

# Funktionale Programmierung

LVA 185.A03, VU 2.0, ECTS 3.0

WS 2019/2020

(Stand: 11.12.2019)

Jens Knoop



Technische Universität Wien  
Information Systems Engineering  
Compilers and Languages



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis

## Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (1)

## Teil I: Einführung

### ► Kap. 1: Motivation

#### 1.1 Ein Beispiel sagt (oft) mehr als 1000 Worte

##### 1.1.1 Zehn Beispiele

##### 1.1.2 Programme auswerten

##### 1.1.3 Programme finden

#### 1.2 Warum funktionale Programmierung? Warum mit Haskell?

##### 1.2.1 Warum funktionale Programmierung?

##### 1.2.2 Imperative vs. funktionale Programmierung

##### 1.2.3 Von imperativer zu funktionaler Programmierung

##### 1.2.4 Funktionale Programmierung: Stärken, Schwächen

##### 1.2.5 Warum funktionale Programmierung mit Haskell?

##### 1.2.6 Erste Schritte in Haskell: Gewöhnliche, literate Haskell-Skripte

#### 1.3 Nützliche Werkzeuge für Haskell: Hugs, GHC, Hoogle, Hayoo, Leksah

#### 1.4 Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (2)

## Teil II: Grundlagen

### ► Kap. 2: Elementare Typen, Tupel, Listen, Zeichenreihen

#### 2.1 Elementare Typen

##### 2.1.1 Wahrheitswerte

##### 2.1.2 Ganze Zahlen

##### 2.1.3 Gleitkommazahlen

##### 2.1.4 Zeichen, Ziffern, Sonderzeichen

#### 2.2 Tupel

#### 2.3 Listen

#### 2.4 Zeichenreihen

#### 2.5 Leseempfehlungen

### ► Kap. 3: Funktionen

#### 3.1 Definition, Schreibweisen, Sprachkonstrukte

#### 3.2 Funktionssignaturen, Funktionsterme, Funktionsstelligkeiten

#### 3.3 Curryfizierte, uncurryfizierte Funktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (3)

- ▶ Kap. 3: Funktionen (fgs.)
  - 3.4 Operatoren, Präfix- und Infixverwendung
  - 3.5 Operatorabschnitte
  - 3.6 Angemessene, unangemessene Funktionsdefinitionen
  - 3.7 Funktions- und Programmlayout, Abseitsregel
  - 3.8 Leseempfehlungen
- ▶ Kap. 4 Typsynonyme, neue Typen, Typklassen
  - 4.1 Typsynonyme (type)
    - 4.1.1 Motivation
    - 4.1.2 Typsynonyme
    - 4.1.3 Tupeltypsynonyme und Selektorfunktionen
    - 4.1.4 Weitere Beispiele
    - 4.1.5 Zusammenfassung
  - 4.2 Neue Typen (newtype)
    - 4.2.1 Motivation
    - 4.2.2 Neue Typen
    - 4.2.3 Typsicherheit erreicht

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (4)

## ► Kap. 4 Typsynonyme, neue Typen, Typklassen (figs.)

### 4.3 Typklassen (class)

#### 4.3.1 Motivation

#### 4.3.2 Vordefinierte Typklassen

#### 4.3.3 Instanzbildung für Typklassen

#### 4.3.4 Automatische Instanzbildung

#### 4.3.5 Selbstdefinierte Typklassen

#### 4.3.6 Zusammenfassung

### 4.4 Leseempfehlungen

## ► Kap. 5: Datentypdeklarationen

### 5.1 Überblick, Orientierung

### 5.2 Algebraische Datentypen (data)

#### 5.2.1 Aufzählungstypen

#### 5.2.2 Produkttypen

#### 5.2.3 Summentypen

#### 5.2.4 Allgemeines Muster

#### 5.2.5 Zusammenfassung

### 5.3 Funktionen auf algebraischen Datentypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (5)

- ▶ Kap. 5: Datentypdeklarationen (fgs.)
  - 5.4 Feldsyntax
  - 5.5 Anwendungshinweise
    - 5.5.1 Produkttypen vs. Tupeltypen
    - 5.5.2 Typsynonyme vs. neue Typen
    - 5.5.3 Faustregel zur Wahl von type, newtype, data
  - 5.6 Leseempfehlungen
- ▶ Kap. 6: Muster und mehr
  - 6.1 Muster, Musterpassung
    - 6.1.1 Muster für Werte elementarer Datentypen
    - 6.1.2 Muster für Werte von Tupeltypen
    - 6.1.3 Muster für Werte von Listentypen
    - 6.1.4 Muster für Werte algebraischer Datentypen
    - 6.1.5 Das als-Muster
    - 6.1.6 Zusammenfassung
  - 6.2 Listenkomprehension
  - 6.3 Konstruktoren, Operatoren
  - 6.4 Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Inhaltsverzeichnis (6)

## Teil III: Applikative Programmierung

### ► Kap. 7: Rekursion

#### 7.1 Motivation

##### 7.1.1 Schnelles Sortieren, Quicksort

##### 7.1.2 Türme von Hanoi

#### 7.2 Rekursionstypen

##### 7.2.1 Mikroskopische Ebene

##### 7.2.2 Makroskopische Ebene

##### 7.2.3 Eleganz und Effizienz, Effizienzfallen

#### 7.3 Aufrufgraphen

#### 7.4 Komplexität, Komplexitätsklassen

#### 7.5 Leseempfehlungen

### ► Kap. 8: Auswertung von Ausdrücken

#### 8.1 Auswertung einfacher Ausdrücke

#### 8.2 Auswertung einfacher funktionaler Ausdrücke

#### 8.3 Zusammenfassung

#### 8.4 Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Inhaltsverzeichnis (7)

## ► Kap. 9: Programmentwicklung, Programmverstehen

9.1 Programmentwicklung

9.2 Programmverstehen

9.3 Leseempfehlungen

## Teil IV Funktionale Programmierung

## ► Kap. 10: Funktionen höherer Ordnung

### 10.1 Motivation

10.1.1 Beispiele vordefinierter Funktionale

10.1.2 Beispiele selbstdefinierter Funktionale

10.1.3 Beispiele aus der Mathematik

10.1.4 Beispiele aus anderen Informatikbereichen

### 10.2 Funktionale Abstraktion

10.2.1 Funktionale Abstraktion 1. Stufe

10.2.2 Funktionale Abstraktion höherer Stufe

10.2.3 Zusammenfassung

### 10.3 Funktionen als Argument

10.3.1 Beispiele

10.3.2 Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (8)

- ▶ Kap. 10: Funktionen höherer Ordnung (fgs.)
  - 10.4 Funktionen als Resultat
    - 10.4.1 Beispiele
    - 10.4.2 Methoden 1 bis 6
    - 10.4.3 Zusammenfassung
  - 10.5 Vordefinierte Funktionale auf Listen
    - 10.5.1 Transformieren: Das Funktional `map`
    - 10.5.2 Filtern: Das Funktional `filter`
    - 10.5.3 Aggregieren: Die Funktionale `foldl`, `foldr`
  - 10.6 Beispiel: Rechnen mit Funktionen
  - 10.7 Zusammenfassung
  - 10.8 Leseempfehlungen
- ▶ Kap. 11: Polymorphie
  - 11.1 Motivation
  - 11.2 Polymorphie auf Datentypen
    - 11.2.1 Polymorphe algebraische Datentypen
    - 11.2.2 Polymorphe neue Typen
    - 11.2.3 Polymorphe Typsynonyme
    - 11.2.4 Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (9)

## ► Kap. 11: Polymorphie (fgs.)

### 11.3 Parametrische Polymorphie auf Funktionen

11.3.1 Vordefinierte parametrisch polymorphe Funktionen

11.3.2 Selbstdefinierte parametrisch polymorphe Funktionen

11.3.3 Zusammenfassung

### 11.4 *Ad hoc* Polymorphie auf Funktionen

11.4.1 Überladene Funktionen vordefinierter Typklassen

11.4.2 Überladene Funktionen selbstdefinierter Typklassen

11.4.3 Vererben, erben, überschreiben

11.4.4 Automatische Typklasseninstanzbildung

11.4.5 Grenzen des Überladens

11.4.6 *Ad hoc* Polymorphie vs. Polymorphie

### 11.5 Zusammenfassung

### 11.6 Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (10)

## Teil V: Fundierung funktionaler Programmierung

### ► Kap. 12: $\lambda$ -Kalkül

#### 12.1 Motivation

#### 12.2 Syntax des reinen $\lambda$ -Kalküls

#### 12.3 Semantik des reinen $\lambda$ -Kalküls

##### 12.3.1 Syntaktische Substitution

##### 12.3.2 Konversionsregeln

##### 12.3.3 Reduktionsfolgen

##### 12.3.4 Normalformen

##### 12.3.5 Semantik von $\lambda$ -Ausdrücken

##### 12.3.6 Rekursion vs. Y-Kombinator

#### 12.4 Angewandte $\lambda$ -Kalküle

#### 12.5 Zusammenfassung

#### 12.6 Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (11)

## ► Kap. 13: Auswertungsordnungen

13.1 Motivation

13.2 Applikative, normale Auswertungsordnung

13.3 Linksapplikative, linksnormale Auswertungsordnung

13.4 Auswertungsordnungscharakterisierungen

13.5 Frühe oder späte Auswertung? Eine Standpunktfrage

13.6 Frühe und späte Auswertung in Haskell

13.7 Namens- und Bezeichnungsbetrachtung

13.8 Leseempfehlungen

## ► Kap. 14: Typprüfung, Typinferenz

14.1 Motivation

14.2 Monomorphe Typprüfung

14.3 Polymorphe Typprüfung

14.3 Polymorphe Typprüfung mit Typklassen

14.5 Typsysteme, Typinferenz

14.6 Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (12)

## Teil VI: Weiterführende Konzepte

### ► Kap. 15: Ein- und Ausgabe

#### 15.1 Motivation

##### 15.1.1 Die Herausforderung

##### 15.1.2 Warum (naive) Einfachheit versagt

#### 15.2 Haskells Lösung

##### 15.2.1 Konzeption und Umsetzung

##### 15.2.2 Aktionen

##### 15.2.3 Aktionssequenzen

##### 15.2.4 Zur Sonderstellung des Typs (IO a)

#### 15.3 E/A-Operationen, E/A-Sequenzen

#### 15.4 Die do-Notation

#### 15.5 Beispiele ausgewählter E/A-Programme

##### 15.5.1 Dialog- und Interaktionsprogramme

##### 15.5.2 Rekursive E/A-Programme

##### 15.5.3 Iterativartige E/A-Programme

##### 15.5.4 'Iteration' vs. Rekursion

##### 15.5.5 Subtiles, Randbemerkung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (13)

- ▶ **Kap. 15: Ein- und Ausgabe (fgs.)**
  - 15.6 Zusammenfassung
  - 15.7 Leseempfehlungen
- ▶ **Kap. 16: Fehlerbehandlung**
  - 16.1 Überblick, Orientierung
  - 16.2 Panikmodus
  - 16.3 Auffangwerte
  - 16.4 Fehlertypen, Fehlerfunktionen
  - 16.5 Leseempfehlungen
- ▶ **Kap. 17: Module**
  - 17.1 Überblick, Orientierung
  - 17.2 Ziele und Richtlinien guter Modularisierung
  - 17.3 Haskells Modulkonzept
    - 17.3.1 Import
    - 17.3.2 Export
    - 17.3.3 Reexport
    - 17.3.4 Namenskonflikte, Umbenennungen, Konventionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (14)

- ▶ Kap. 17: Module (fgs.)
  - 17.4 Abstrakte Datentypen als Modul-Anwendung
  - 17.5 Leseempfehlungen
- ▶ Kap. 18: Programmierprinzipien
  - 18.1 Überblick, Orientierung
  - 18.2 Teile und Herrsche
  - 18.3 Stromprogrammierung
  - 18.4 Reflektives Programmieren
  - 18.5 Leseempfehlungen

## Teil VII: Abschluss, Ausblick

- ▶ Kap. 19: Abschließendes
  - 19.1 Rückblick
  - 19.2 Ausblick
  - 19.3 Leseempfehlungen
- ▶ Literaturverzeichnis



# Inhaltsverzeichnis (15)

## Anhänge

- ▶ A Imperative vs. funktionale Programmierung: Schlaglichter
  - A.1 Programmatischer Kern
  - A.2 Namensvereinbarungen
  - A.3 Operanden und Werte von Ausdrücken
  - A.4 Funktionen und Polymorphie: Erstrangige Sprachelemente
  - A.5 Imperative vs. funktionale Programme
  - A.6 Wertzuweisung vs. Wertvereinbarung
  - A.7 Selbstbezügliche Wertzuweisungen, Wertvereinbarungen
  - A.8 Problem- und Lösungssicht: Imperativ vs. funktional
  - A.9 Welcher Problemlösungstyp bin ich?
  - A.10 Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Inhaltsverzeichnis (16)

- ▶ B Formale Rechenmodelle
  - B.1 Turing-Maschinen
  - B.2 Markov-Algorithmen
  - B.3 Primitiv-rekursive Funktionen
  - B.4  $\mu$ -rekursive Funktionen
  - B.5 Leseempfehlungen
- ▶ C Andere funktionale Sprachen
- ▶ D Datentypdeklarationen in Pascal
- ▶ E Implementierungsaspekte
- ▶ F Hinweise zur schriftlichen Prüfung

## Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Teil I

## Einführung

Inhalt

**Teil I**

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 1

## Motivation

Inhalt

Teil I

**Kap. 1**

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Das leere Haskell-Programm

Inhalt

Teil I

**Kap. 1**

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Das leere Haskell-Programm: Mehr als nichts!

...bereits das [leere Haskell-Programm](#) bietet

## Taschenrechnerfunktionalität:

```
>hugs
```

```
Main>:load leeresHaskellProgramm.hs
```

```
Main>2+3
```

```
5
```

```
Main>abs (5-12)
```

```
7
```

```
Main>sqrt 121
```

```
11.0
```

```
Main>abs (-5) * 6 + 3 <= 2^3 * (4 + round 3.14)
```

```
True
```

```
Main>sin 0
```

```
0.0
```

```
Main>cos 0
```

```
1.0
```

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Das leere Haskell-Programm: Mehr als nichts!

...und mehr:

```
Main>True && False
```

```
False
```

```
Main>not (True && False)
```

```
True
```

```
Main>"Funktionale" ++ " " ++ "Programmierung"
```

```
"Funktionale Programmierung"
```

```
Main>length "Funktionale Programmierung"
```

```
26
```

```
Main>[1..12]
```

```
[1,2,3,4,5,6,7,8,9,10,11,12]
```

```
Main>[1,4..12]
```

```
[1,4,7,10]
```

```
Main>length [10..20]
```

```
11
```

```
Main>[n | n <- [-6..8], mod n 2 == 0]
```

```
[-6,-4,-2,0,2,4,6,8]
```

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-23/1707

# Überblick

## Funktionale Programmierung, funktionale Programmierung in Haskell

- 1.1 Ein Beispiel sagt (oft) mehr als 1000 Worte
- 1.2 Warum funktionale Programmierung? Warum mit Haskell?
- 1.3 Nützliche Werkzeuge für Haskell: Hugs, GHC, GHCi, Hoogle, Hayoo, Leksah
- 1.4 Literaturverzeichnis, Leseempfehlungen

**Anmerkung:** Einige Begriffe werden in diesem Kapitel im Vorgriff angerissen und erst im Lauf der Vorlesung genau geklärt!

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-24/1707



# Kapitel 1.1

Ein Beispiel sagt (oft) mehr als 1000 Worte

Inhalt

Teil I

Kap. 1

**1.1**

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Kapitel 1.1.1

## Zehn Beispiele

Inhalt

Teil I

Kap. 1

1.1

**1.1.1**

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Zehn Beispiele

Longum iter est per praecepta, breve et efficax per exempla.

Lang ist der Weg über Belehrungen,  
kurz und wirkungsvoll durch Beispiele.

Seneca der Jüngere (um 4 v.Chr. - 65 n.Chr.)  
röm. Politiker, Philosoph und Schriftsteller

1. *Hello, World!*
2. Fakultätsfunktion
3. Das Sieb des Eratosthenes
4. Binomialkoeffizienten
5. Umkehren einer Zeichenreihe
6. Reißverschlussfunktion
7. Addition
8. Map-Funktion
9. Euklidischer Algorithmus
10. Gerade/ungerade-Test

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

27/1707

# 1) Hello, World!

```
main = putStrLn "Hello, World!"
```

...ein Beispiel für ein Programm mit **Ein-/Ausgabeoperation**.

**Nicht selbsterklärend:** Die Deklaration von `putStrLn`

```
putStrLn :: String -> IO ()  
putStrLn "Hello, World!"
```

**Allerdings:** Die **Java**-Entsprechung

```
class HelloWorld {  
    public static void main (String[] args) {  
        System.out.println("Hello, World!"); } }  
}
```

...bedarf auch einer weiter ausholenden Erläuterung.

## 2) Fakultätsfunktion (1)

$$! : \mathbb{IN} \rightarrow \mathbb{IN}$$

$$\forall n \in \mathbb{IN}. n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n - 1)! & \text{sonst} \end{cases}$$

`fac :: Integer -> Integer`

`fac n = if n == 0 then 1 else n * fac (n - 1)`

...ein Beispiel für eine **rekursive** Funktionsdefinition.

**Aufrufe:**

`fac 0 ->> 1`    `fac 3 ->> 6`    `fac 6 ->> 720`  
`fac 1 ->> 1`    `fac 5 ->> 120`    `fac 10 ->> 3.628.800`

**Lies:** “Die Auswertung des Ausdrucks/Aufrufs `fac 5` liefert den Wert `120`; der Ausdruck/Aufruf `fac 5` hat den Wert `120`.”

## 2) Fakultätsfunktion (2)

```
fac :: Integer -> Integer
```

```
fac n = if n == 0 then 1 else n * fac (n - 1)
```

Funktionale Programmierung mag es **kurz und knackig**, **prägnant und konzis**, ohne **kryptisch** zu sein. Auch **Haskell** hat hierfür ein Angebot.

### Alternative Schreibweise:

```
fac :: Integer -> Integer
```

```
fac n
```

```
  | n == 0      = 1                (| für (oder) wenn)
```

```
  | otherwise = n * fac (n - 1)    (otherwise -> True)
```

```
fac :: Integer -> Integer          (Diese Variante nur zur  
fac n                             Illustration von |)
```

```
  | n == 0 || n == 1 = 1          ((||) logisches oder)
```

```
  | n == 2           = 2
```

```
  | otherwise        = n * fac (n - 1)
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

30/1707

## 2) Fakultätsfunktion (3)

Eine zweite weitere Schreibweise, musterbasiert:

```
fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n - 1)
```

Weitere alternative Implementierungen:

```
fac :: Integer -> Integer
fac n = foldl (*) 1 [1..n]      ([1..3] ->> [1,2,3],
                                [1..0] ->> [],
                                foldl (*) 1 [1..0] ->> foldl (*) 1 [] ->> 1)
```

```
fac :: Integer -> Integer
fac n = product [1..n]         (product = foldl (*) 1)
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

31/1707

## 2) Fakultätsfunktion (4)

Zwei einfache Formen der Fehlerbehandlung:

```
fac' :: Integer -> Integer
```

```
fac' n
```

```
  | n == 0      = 1
```

```
  | n > 0       = n * fac' (n - 1)
```

```
  | otherwise = error "Arg. unzulässig" (sog. Panikmodus)
```

```
fac'' :: Integer -> Integer
```

```
fac'' n
```

```
  | n == 0      = 1
```

```
  | n > 0       = n * fac'' (n - 1)
```

```
  | otherwise = n (Sonderwertrückgabe)
```

Aufrufe:

```
fac' 5    ->> 120
```

```
fac' 0    ->> 1
```

```
fac' (-5) ->> "Arg. unzulässig"
```

```
fac'' 5    ->> 120
```

```
fac'' 0    ->> 1
```

```
fac'' (-5) ->> -5
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

32/1707



# Funktional vs. imperativ: Kurzer Exkurs (1)

Vergleiche folgende **funktionale** und **imperative** Implementierungen der Fakultätsfunktion:

**Funktional, hier in Haskell:**

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else n * fac (n-1)
```

**Imperativ, hier in Pascal:**

```
FUNCTION fac (n: integer): integer;
BEGIN
  IF n=0 THEN fac := 1 ELSE fac := n*fac(n-1)
END;
```

**Beachte:** Trotz der äußerlichen Ähnlichkeit sind die **funktionale** und **imperative** **Fallunterscheidung**, die in beiden Fällen die Fakultätsfunktion definieren, **konzeptuell äußerst verschieden!**

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

33/1707

# Funktional vs. imperativ: Kurzer Exkurs (2)

Die Fallunterscheidung 'if-then-else' im Vergleich:

Funktional:

```
fac :: Integer -> Integer
```

```
fac n = if n == 0 then 1 else n * fac (n-1)
```

*Ausdruck*   *Ausdruck*   *Ausdruck*

*Ausdruck*

Imperativ:

```
FUNCTION fac (n: integer): integer;
```

```
BEGIN IF n=0 THEN fac:=1 ELSE fac:=n*fac(n-1) END;
```

*Ausdruck*   *Ausdruck*   *Ausdruck*

*Anweisung*   *Anweisung*

*Anweisung*

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

34/1707

# Funktional vs. imperativ: Kurzer Exkurs (3)

## Die Fallunterscheidung 'if-then-else':

- ▶ **Funktional:** Die Fallunterscheidung ist ein **Ausdruck**. Ihre Bedeutung (Semantik) ist ein **Wert**.
- ▶ **Imperativ:** Die Fallunterscheidung ist eine **Anweisung**. Ihre Bedeutung (Semantik) ist eine **Zustandstransformation**, eine Belegung von Variablen mit (neuen) Werten.

Dieser Unterschied in Konzept u. Bedeutung ist fundamental.

- ▶ 'if-then-else' funktional  $\neq$  'if-then-else' imperativ

...es ist **wichtig**, sich diesen **Unterschied klarzumachen** und zu **verinnerlichen** (s.a. Kap. 1.2.2, Kap. 1.2.3, Anh. A.1).

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

35/1707

### 3) Das Sieb des Eratosthenes (276 - 194 v.Chr.)

...zur Berechnung der **unendlichen Folge** der **Primzahlen**:

1. Schreibe **alle** natürlichen Zahlen ab **2** hintereinander auf.
2. Die kleinste nicht gestrichene Zahl in dieser Folge ist eine **Primzahl**. Streiche alle Vielfachen dieser Zahl.
3. Wiederhole Schritt 2 mit der kleinsten jeweils noch nicht gestrichenen Zahl.

Nach Schritt 1:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19...

Nach Schritt 2 für Zahl 2:

2 3 5 7 9 11 13 15 17 19...

Nach Schritt 2 für Zahl 3:

2 3 5 7 11 13 17 19...

usw.

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

### 3) Das Sieb des Eratosthenes (2)

`primes :: [Integer]`  
*Zahlenstromtyp* (primes, der (Prim-) Zahlenstrom als Integer-Liste)

`primes = sieve [2..]`  
*Strom der nat. Zahlen ab 2* (leistet Schritt 1)

`sieve :: [Integer] -> [Integer]`  
*Argumentstromtyp* *Resultatstromtyp*

`sieve (x:xs) = x : sieve [y | y <- xs, mod y x > 0]`  
*Argumentstrom* *Resultatstrom*  
(leistet Schritt 2 für die Zahlen 2, 3, 5, 7, 11, usw.)

...ein Beispiel für die Programmierung mit **Strömen**.

### 3) Das Sieb des Eratosthenes (3)

`primes :: [Integer]`  
*(Prim-) Zahlenstromtyp*

`sieve :: [Integer]`  $\rightarrow$  `[Integer]`  
*Argumentstromtyp* *Resultatstromtyp*

`sieve (x:xs)` = `x : sieve [y | y <- xs, mod y x > 0]`  
*Strom der nat. Zahlen ab 2 als Argument* *Strom der Primzahlen als Resultat*

`primes = sieve [2..]`  
*Strom der nat. Zahlen ab 2*  
*Strom der Primzahlen*

Aufruf:

`primes ->> sieve [2..] ->> 2 : sieve [3,5..]`  
`->> ... ->> [2,3,5,7,11,13,17,19,23,29,31,37,41,...]`

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

38/1707

### 3) Das Sieb des Eratosthenes (4)

Im Überblick und (fast) ohne Farbspiele:

```
primes :: [Integer]
primes = sieve [2..]

sieve :: [Integer] -> [Integer]
sieve (x:xs) = x : sieve [y | y <- xs, mod y x > 0]
```

Aufrufe:

```
primes ->> [2,3,5,7,11,13,17,19,23,29,31,37,41,...]
take 5 primes ->> [2,3,5,7,11]
drop 3 primes ->> [7,11,13,17,19,23,29,31,37,41,...]
take 3 (drop 3 primes) ->> [7,11,13]

primes!!0 ->> 2                ((!!) Zugriffsoperator)
primes!!1 ->> 3
primes!!5 ->> 13
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

39/1707

## 4) Binomialkoeffizienten (1)

...geben die Anzahl der Kombinationen  $k$ -ter Ordnung von  $n$  Elementen ohne Wiederholung an:

$$(\cdot) : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall n, k \in \mathbb{N}. \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

```
binom' :: (Integer,Integer) -> Integer
binom' (n,k) = div (fac n) (fac k * fac (n-k))
```

...ein Beispiel für eine **musterbasierte** Funktionsdefinition mit **hierarchischer Abstützung** auf eine andere Funktion ('Hilfsfunktion'), hier die Fakultätsfunktion.

Aufrufe:

```
binom' (49,6) ->> 13.983.816
binom' (45,6) ->> 8.145.060
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

40/1707



## 4) Binomialkoeffizienten (2)

Es gilt:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

`binom'' :: (Integer,Integer) -> Integer`

`binom'' (n,k)`

`| k==0 || n==k = 1`

`| otherwise = binom'' (n-1,k-1) + binom'' (n-1,k)`

...ein Beispiel für eine **musterbasierte (kaskaden- oder baum-artig-) rekursive** Funktionsdefinition.

Aufrufe:

`binom'' (49,6) ->> 13.983.816`

`binom'' (45,6) ->> 8.145.060`

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

## 4) Binomialkoeffizienten (3)

### Uncurryfiziert

```
binom' :: (Integer,Integer) -> Integer  
binom' (n,k) = div (fac n) (fac k * fac (n-k))
```

### Curryfiziert

```
binom :: Integer -> (Integer -> Integer)  
binom n k = div (fac n) (fac k * fac (n-k))
```

### Aufrufe:

```
binom' (49,6) ->> 13.983.816  
binom' (45,6) ->> 8.145.060  
binom 49 6 ->> 13.983.816  
binom 45 6 ->> 8.145.060  
binom 49  
binom 45 ...sind ebenfalls zulässige Ausdrücke!
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

42/1707

## 4) Binomialkoeffizienten (4)

Die Aufrufe

```
binom 49
```

```
binom 45
```

...sind gültige Ausdrücke von einem funktionalen Wert:

```
(binom 49) :: Integer -> Integer
```

```
(binom 45) :: Integer -> Integer
```

...und repräsentieren die Funktionen '49\_über\_k' (entsprechend 'k\_aus\_49') und '45\_über\_k' (entsprechend 'k\_aus\_45'):

$$\binom{49}{\cdot} : \mathbb{N} \rightarrow \mathbb{N}$$

$$\binom{45}{\cdot} : \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall k \in \mathbb{N}. \binom{49}{k} = \frac{49!}{k!(49-k)!} \quad \forall k \in \mathbb{N}. \binom{45}{k} = \frac{45!}{k!(45-k)!}$$

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

43/1707

## 4) Binomialkoeffizienten (5)

In der Tat können wir als Funktionen definieren:

```
k_aus_49 :: Integer -> Integer
```

```
k_aus_49 k = binom 49 k
```

```
k_aus_45 :: Integer -> Integer
```

```
k_aus_45 k = binom 45 k
```

...und punktfrei (d.h., argumentlos) noch knapper:

```
k_aus_49 :: Integer -> Integer
```

```
k_aus_49 = binom 49
```

```
k_aus_45 :: Integer -> Integer
```

```
k_aus_45 = binom 45
```

Aufrufe:

```
k_aus_49 6 ->> binom 49 6 ->> 13.983.816
```

```
k_aus_45 6 ->> binom 45 6 ->> 8.145.060
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

44/1707

## 5) Umkehren einer Zeichenreihe

`data [a] = [] | a:[a]` (Informelle algebraische  
Datentypspez. für Listen)

`type String = [Char]` (Typsynonym)

`reverse :: String -> String`

`reverse "" = ""` ("`""` == [], leere Zeichenreihe)

`reverse (c:cs) = (reverse cs) ++ [c]`

...ein Beispiel für eine Funktion auf [Zeichenreihen](#).

[Aufrufe:](#)

`reverse "" ->> ""`

`reverse "stressed" ->> "desserts"`

`reverse "desserts" ->> "stressed"`

## 6) Reißverschlussfunktion

...zum 'Verschmelzen' zweier Listen zu einer Liste von Paaren:

```
zip :: [a] -> [b] -> [(a,b)]   (a, b Typvariablen)
zip _ []                  = []      (_ sog. "wild card")
zip [] _                  = []      ([] leere Liste)
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

*Liste mit Kopf y und Rest ys  
: sog. Listenkonstruktor*

...ein Beispiel für eine **polymorphe** Funktion auf **Listen**.

Aufrufe:

```
zip [2,3,5,7] ['a','b'] ->> [(2,'a'),(3,'b')]
zip [] ["stressed","desserts"] ->> []
zip [1.1,2.2.3.3] ["fun"] ->> [(1.1,"fun")]
zip [1.1,2.2.3.3] "fun"
->> zip [1.1,2.2.3.3] ['f','u','n']
->> [(1.1,'f'),(2.2,'u'),(3.3,'n')]
```

## 7) Addition

`(+)` :: Num a => a -> a -> a (Num sog. Typklasse)

...ein Beispiel für eine überladene Funktion.

Aufrufe:

`(+)` 2 3 ->> 5 (+ auf ganzen Z., Präfixop.)

2 + 3 ->> 5 (+ als Infixop. auf g.Z.)

`(+)` 2.1 1.4 ->> 3.5 (+ auf Gleitkommaz., Präfixop.)

2.1 + 1.4 ->> 3.5 (+ als Infixop. auf Gkz)

`(+)` 7.81 2 ->> 9.81 (automatische Typanpassung)

`((+) 1)` :: Integer -> Integer (Inkrementfunktion)

`inc` :: Integer -> Integer

`inc = (+) 1` (vgl. die Funktion '(binom 49)')

`inc'` :: Integer -> Integer

`inc' = (+1)` ((+1) ein sog. Operatorabschnitt)

## 8) Die map-Funktion

...zur Anwendung einer Funktion auf alle Elemente einer Liste:

```
map :: (a -> b) -> [a] -> [b]           (Fkt. als Arg.)  
map _ [] = []  
map f (x:xs) = (f x) : map f xs
```

...ein Beispiel für eine **Funktion höherer Ordnung**, für Funktionen als **erstrangige Sprachelemente** (engl. **first class citizens**).

Aufrufe:

```
map (2*) [1,2,3,4,5] ->> [2,4,6,8,10]  
map (\x -> x*x) [1,2,3,4,5] ->> [1,4,9,16,25]  
map (>3) [2,3,4,5] ->> [False,False,True,True]  
map length ["functional","programming","is","fun"]  
           ->> [10,11,2,3]
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

48/1707



# 9) Der Euklidische Algorithmus (3.Jhdt.v.Chr.)

...zur Berechnung des **größten gemeinsamen Teilers** zweier natürlicher Zahlen  $m$ ,  $n$  mit  $m \geq 0$  und  $n > 0$ :

**ggT** :: Int -> Int -> Int (Ganzz.-Typ, beschränkt)

**ggT** m n

| n == 0 = m

| n > 0 = **ggT** n (mod m n)

**mod** :: Int -> Int -> Int

**mod** m n

| m < n = m

| m >= n = **mod** (m-n) n

...ein Beispiel für ein **hierarchisches System von Funktionen**.

Aufrufe:

**ggT** 25 15 ->> 5    **ggT** 48 60 ->> 12    **mod** 8 3 ->> 2

**ggT** 28 60 ->> 4    **ggT** 60 40 ->> 20    **mod** 9 3 ->> 0

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

49/1707

## 10) Gerade/ungerade-Test

```
isEven :: Integer -> Bool
```

```
isEven n
```

```
  | n == 0 = True
```

```
  | n > 0  = isOdd (n-1)
```

```
isOdd :: Integer -> Bool
```

```
isOdd n
```

```
  | n == 0 = False
```

```
  | n > 0  = isEven (n-1)
```

...ein Beispiel für ein **System wechselseitig** (oder **indirekt**) **rekursiver Funktionen**.

Aufrufe:

```
isEven 6 ->> True
```

```
isOdd 6 ->> False
```

```
isEven (-5) ->> Musterfehler
```

```
isEven 9 ->> False
```

```
isOdd 9 ->> True
```

```
isOdd (-1) ->> Musterfehler
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

50/1707

# Im Rückblick: Die ersten zehn Beispiele (1)

## 1. Ein- und Ausgabe

- ▶ *Hello, World!*

## 2. Rekursion

- ▶ Fakultätsfunktion

## 3. Stromprogrammierung

- ▶ Das Sieb des Eratosthenes

## 4. Musterbasierte, curryfizierte und uncurryfizierte Funktionsdefinitionen, partiell ausgewertete Funktionen

- ▶ Binomialkoeffizienten

## 5. Funktionen auf Zeichenreihen

- ▶ Umkehren einer Zeichenreihe

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

# Im Rückblick: Die ersten zehn Beispiele (2)

6. Parametrisch polymorphe Funktionen
  - ▶ Reißverschlussfunktion
7. Überladene Funktionen
  - ▶ Addition
8. Fkt. höherer Ordnung, Fkt. als 'Bürger erster Klasse'
  - ▶ Map-Funktion
9. Hierarchische Systeme von Funktionen
  - ▶ Euklidischer Algorithmus
10. Systeme wechselseitig rekursiver Funktionen
  - ▶ Gerade/ungerade-Test

Nichts ist schwerer zu befolgen  
als ein gutes Beispiel.

Mark Twain (1835-1910)  
amerik. Schriftsteller

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

52/1707

# Wir halten fest

Funktionale Programme sind

- ▶ Systeme (wechselweise) rekursiver Funktionsvorschriften (oder Rechenvorschriften).

Funktionen

- ▶ sind zentrales Abstraktionsmittel in funktionalen Programmen (wie Prozeduren (Methoden) in prozeduralen (objektorientierten) Programmen).

Funktionale Programme

- ▶ werten Ausdrücke aus. Das Resultat dieser Auswertung ist ein Wert eines bestimmten Typs. Dieser Wert kann elementar oder funktional sein; er ist die Bedeutung, die Semantik des Ausdrucks.

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

53/1707

# Übungsaufgabe 1.1.1.1

1. Reimplementieren Sie die 10 Beispielprogramme aus Kapitel 1.1.1 in einer Programmiersprache Ihrer Wahl verschieden von Haskell, z.B. in Java, und vergleichen Sie die Programme miteinander, z.B. im Hinblick auf ihre Länge. Scheinen Ihnen die Programme der von Ihnen gewählten Sprache programmiertechnisch aufwändiger oder weniger aufwändig, einfacher oder weniger einfach verständlich? Wenn ja, warum?
2. Kehren Sie zu obigen Fragen im Lauf des Semesters zurück, speziell am Ende. Überlegen Sie dann, ob die heutigen Gründe für Ihre Einschätzung noch gültig sind. Wenn nein, warum nicht?

# Kapitel 1.1.2

## Programme auswerten

Inhalt

Teil I

Kap. 1

1.1

1.1.1

**1.1.2**

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Auswerten einfacher Ausdrücke (1)

Der Ausdruck  $(15*7 + 12) * (7 + 15*12)$  hat den Wert 21.879; seine Semantik ist der Wert 21.879.

```
(15*7 + 12) * (7 + 15*12)
->> (105 + 12) * (7 + 180)
->> 117 * 187
->> 21.879
```

Auch andere Auswertungsreihenfolgen sind möglich, z.B.:

```
(15*7 + 12) * (7 + 15*12)
->> (105 + 12) * (7 + 180)
->> 105*7 + 105*180 + 12*7 + 12*180
->> 735 + 18.900 + 84 + 2.160
->> 21.879
```



# Auswerten einfacher Ausdrücke (2)

$$(15*7 + 12) * (7 + 15*12)$$

$$\rightarrow (105 + 12) * (7 + 180)$$

$$\rightarrow 117 * (7 + 180)$$

$$\rightarrow 117*7 + 117*180$$

$$\rightarrow 819 + 21.060$$

$$\rightarrow 21.879$$

...und viele mehr. Stets ist das Ergebnis gleich!  
(s. Kap. 12.3.4, Church-Rosser-Theoreme).

Die einzelnen Vereinfachungs-, Rechenschritte nennen wir  
► Simplifikationen.

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

57/1707

# Auswerten von Funktionsaufrufen (1)

Der Aufruf `fac 2` hat den Wert 2; seine Semantik (oder Bedeutung) ist der Wert 2.

Eine erste Auswertungsreihenfolge:

`fac 2`

```
(E) ->> if 2 == 0 then 1 else (2 * fac (2-1))
(S) ->> if False then 1 else (2 * fac (2-1))
(S) ->> 2 * fac (2-1)
(S) ->> 2 * fac 1
(E) ->> 2 * (if 1 == 0 then 1 else (1 * fac (1-1)))
(S) ->> 2 * (if False then 1 else (1 * fac (1-1)))
(S) ->> 2 * (1 * fac (1-1))
(S) ->> 2 * (1 * fac 0)
(E) ->> 2 * (1 * (if 0 == 0 then 1 else (0 * fac (0-1))))
(S) ->> 2 * (1 * (if True then 1 else (0 * fac (0-1))))
(S) ->> 2 * (1 * (1))
(S) ->> 2 * (1 * 1)
(S) ->> 2 * 1
(S) ->> 2
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

58/1707

# Auswerten von Funktionsaufrufen (2)

Eine zweite Auswertungsreihenfolge:

```
        fac 2
(E) ->> if 2 == 0 then 1 else (2 * fac (2-1))
(S) ->> if False then 1 else (2 * fac (2-1))
(S) ->> 2 * fac (2-1)
(E) ->> 2 * (if (2-1) == 0 then 1
              else ((2-1) * fac ((2-1)-1)))
(S) ->> 2 * (if 1 == 0 then 1
              else ((2-1) * fac ((2-1)-1)))
(S) ->> 2 * (if False then 1
              else ((2-1) * fac ((2-1)-1)))
(S) ->> 2 * ((2-1) * fac ((2-1)-1))
(S) ->> 2 * (1 * fac ((2-1)-1))
(E) ->> 2 * (1 * (if ((2-1)-1) == 0 then 1
                    else (((2-1)-1) * fac (((2-1)-1))))
(S) ->> 2 * (1 * (if (1-1) == 0 then 1
                    else (((2-1)-1) * fac (((2-1)-1))))
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Auswerten von Funktionsaufrufen (3)

```
(S) ->> 2 * (1 * (if 0 == 0 then 1
                    else (((2-1)-1) * fac (((2-1)-1))))
(S) ->> 2 * (1 * (if True then 1
                    else (((2-1)-1) * fac (((2-1)-1))))
(S) ->> 2 * (1 * 1)
(S) ->> 2 * 1
(S) ->> 2
```

Wir bezeichnen mit

- ▶ (E) markierte Schritte als **Expansionsschritte**.
- ▶ (S) markierte Schritte als **Simplifikationsschritte**.

# Auswerten von Funktionsaufrufen (4)

Die beiden **Auswertungsreihenfolgen** sind Beispiele sog.

- ▶ **applikativer** (**sofortiger**, **unverzögerlicher**) (1. Ausw.folge, z.B. in **ML**)
- ▶ **normaler** (**aufgeschobener**, **verzögerter**) (2. Ausw.folge, z.B. in **Haskell**)

**Auswertung.**

# Applikative Auswertung des Aufrufs natSum 2

Berechnung von  $\sum_{i=0}^n i$  für positive Werte  $n \in \mathbb{N}_0$  durch `natSum`:

```
natSum :: Int -> Int
```

```
natSum n = if n == 0 then 0 else (natSum (n-1)) + n
```

```
natSum 2
```

```
(E) ->> if 2 == 0 then 0 else (natSum (2-1)) + 2
```

```
(S) ->> if False then 0 else (natSum (2-1)) + 2
```

```
(S) ->> (natSum (2-1)) + 2
```

```
(S) ->> (natSum 1) + 2
```

```
(E) ->> (if 1 == 0 then 0 else ((natSum (1-1)) + 1)) + 2
```

```
(S) ->> (if False then 0 else ((natSum (1-1)) + 1)) + 2
```

```
(S) ->> ((natSum (1-1)) + 1) + 2
```

```
(S) ->> ((natSum 0) + 1) + 2
```

```
(E) ->> ((if 0 == 0 then 0 else (natSum (0-1)) + 0) + 1) + 2
```

```
(E) ->> ((if True then 0 else (natSum (0-1)) + 0) + 1) + 2
```

```
(S) ->> ((0) + 1) + 2
```

```
(S) ->> (0 + 1) + 2
```

```
(S) ->> 1 + 2
```

```
(S) ->> 3
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

62/1707

# Normale Auswertung des Aufrufs natSum 2

```
natSum 2
(E) ->> if 2 == 0 then 0 else (natSum (2-1)) + 2
(S) ->> if False then 0 else (natSum (2-1)) + 2
(S) ->> (natSum (2-1)) + 2
(E) ->> if (2-1) == 0 then 0 else (natSum ((2-1)-1)) + (2-1) + 2
(S) ->> if 1 == 0 then 0 else (natSum ((2-1)-1)) + (2-1) + 2
(S) ->> if False then 0 else (natSum ((2-1)-1)) + (2-1) + 2
(S) ->> (natSum ((2-1)-1)) + (2-1) + 2
(E) ->> ...
...
(S) ->> 3
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

63/1707

# Übungsaufgabe 1.1.2.1

Vervollständige die normale Auswertung des Aufrufs `natSum 2`.

Inhalt

Teil I

Kap. 1

1.1

1.1.1

**1.1.2**

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV



# Kapitel 1.1.3

## Programme finden

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

**1.1.3**

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

65/1707

# 'Finden' rekursiver Formulierungen (1)

...am Beispiel der **Fakultätsfunktion**:

`fac n = n*(n-1)*...*6*5*4*3*2*1*1`

Von der Lösung erwarten wir also:

`fac 0 = 1 ->> 1`

`fac 1 = 1*1 ->> 1`

`fac 2 = 2*1*1 ->> 2`

`fac 3 = 3*2*1*1 ->> 6`

`fac 4 = 4*3*2*1*1 ->> 24`

`fac 5 = 5*4*3*2*1*1 ->> 120`

`fac 6 = 6*5*4*3*2*1*1 ->> 720`

...

`fac n = n*(n-1)*...*6*5*4*3*2*1*1 ->> n!`

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

66/1707

# 'Finden' rekursiver Formulierungen (2)

$$\text{fac } 0 = 1 \quad \rightarrow 1$$

$$\text{fac } 1 = 1 * \underbrace{1}_{\text{fac } 0} \quad == 1 * \text{fac } 0 \rightarrow 1$$

$$\text{fac } 2 = 2 * \underbrace{1*1}_{\text{fac } 1} \quad == 2 * \text{fac } 1 \rightarrow 2$$

$$\text{fac } 3 = 3 * \underbrace{2*1*1}_{\text{fac } 2} \quad == 3 * \text{fac } 2 \rightarrow 6$$

$$\text{fac } 4 = 4 * \underbrace{3*2*1*1}_{\text{fac } 3} \quad == 4 * \text{fac } 3 \rightarrow 24$$

$$\text{fac } 5 = 5 * \underbrace{4*3*2*1*1}_{\text{fac } 4} \quad == 5 * \text{fac } 4 \rightarrow 120$$

...

$$\text{fac } n = n * \underbrace{(n-1)*\dots*6*5*4*3*2*1*1}_{\text{fac } (n-1)}$$

$$== n * \text{fac } (n-1) \rightarrow n!$$

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

67/1707

# ‘Finden’ rekursiver Formulierungen (3)

Wir erkennen:

- ▶ Ein Sonderfall:  $\text{fac } 0 = 1$
- ▶ Ein Regelfall:  $\text{fac } n = n * \text{fac } (n-1)$

Wir führen beide Fälle zusammen und erhalten:

```
fac n
| n == 0      = 1
| otherwise = n * fac (n-1)
```

Noch unmittelbarer zusammengeführt:

```
fac 0 = 1
fac n = n * fac (n-1)
```

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

68/1707

# ‘Finden’ rekursiver Formulierungen (4)

...am Beispiel der Berechnung von  $0+1+2+3+\dots+n$ :

$$\text{natSum } n = 0+1+2+3+4+5+6+\dots+(n-1)+n$$

Von der Lösung erwarten wir also:

$$\text{natSum } 0 = 0 \rightarrow 0$$

$$\text{natSum } 1 = 0+1 \rightarrow 1$$

$$\text{natSum } 2 = 0+1+2 \rightarrow 3$$

$$\text{natSum } 3 = 0+1+2+3 \rightarrow 6$$

$$\text{natSum } 4 = 0+1+2+3+4 \rightarrow 10$$

$$\text{natSum } 5 = 0+1+2+3+4+5 \rightarrow 15$$

$$\text{natSum } 6 = 0+1+2+3+4+5+6 \rightarrow 21$$

...

$$\text{natSum } n = 0+1+2+3+4+5+6+\dots+(n-1)+n$$

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

69/1707

# 'Finden' rekursiver Formulierungen (5)

$$\text{natSum } 0 = 0 \quad \rightarrow 0$$

$$\text{natSum } 1 = \underbrace{0 + 1}_{\text{natSum } 0} \quad == \text{natSum } 0 + 1 \rightarrow 1$$

$$\text{natSum } 2 = \underbrace{0+1}_{\text{natSum } 1} + 2 \quad == \text{natSum } 1 + 2 \rightarrow 3$$

$$\text{natSum } 3 = \underbrace{0+1+2}_{\text{natSum } 2} + 3 \quad == \text{natSum } 2 + 3 \rightarrow 6$$

$$\text{natSum } 4 = \underbrace{0+1+2+3}_{\text{natSum } 3} + 4 \quad == \text{natSum } 3 + 4 \rightarrow 10$$

$$\text{natSum } 5 = \underbrace{0+1+2+3+4}_{\text{natSum } 4} + 5 \quad == \text{natSum } 4 + 5 \rightarrow 15$$

...

$$\begin{aligned} \text{natSum } n &= \underbrace{0+1+2+3+4+5+6+\dots+(n-1)}_{\text{natSum } (n-1)} + n \\ &== \text{natSum } (n-1) + n \rightarrow \text{natSum } n \end{aligned}$$

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

70/1707

# 'Finden' rekursiver Formulierungen (6)

Wir erkennen:

- ▶ Ein Sonderfall: `natSum 0 = 0`
- ▶ Ein Regelfall: `natSum n = (natSum (n-1)) + n`

Wir führen beide Fälle zusammen und erhalten:

```
natSum n
| n == 0      = 0
| otherwise = (natSum (n-1)) + n
```

Noch unmittelbarer zusammengeführt:

```
natSum 0 = 0
natSum n = (natSum (n-1)) + n
```

Anm.: Einfachere Klammerung ist möglich: `natSum (n-1) + n`

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

1.1.3

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

71/1707

Est omnium rerum magister usus.  
Aller Dinge Meister ist der Gebrauch.

Julius Caesar (100 - 44 v.Chr.)  
röm. Feldherr und Staatsmann

Inhalt

Teil I

Kap. 1

1.1

1.1.1

1.1.2

**1.1.3**

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV



# Kapitel 1.2

Warum funktionale Programmierung?  
Warum mit Haskell?

Inhalt

Teil I

Kap. 1

1.1

**1.2**

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 1.2.1

## Warum funktionale Programmierung?

Res loquitur ipsa.

Die Sache spricht für sich.

Cicero (106 - 43 v.Chr.)

röm. Staatsmann und Schriftsteller

For at least the last 60 years, programmers  
have been faced with the question:  
What programming language should I use?  
Myriad languages have been developed  
in the last six decades, with at least a few  
dozen in common usage today.

Jeffrey S. Foster

*Shedding new Light on an old Language Debate.*  
Communications of the ACM 60(10):90, 2017.

# Etwas spezieller die Frage von John W. Backus

*Can programming be liberated  
from the von Neumann style?*

John W. Backus (1924-2007)  
*Turing Award* Preisträger 1977

John W. Backus. *Can Programming be Liberated from the von  
Neumann Style? A Functional Style and its Algebra of Programs.*  
Communications of the ACM 21(8):613-641, 1978.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

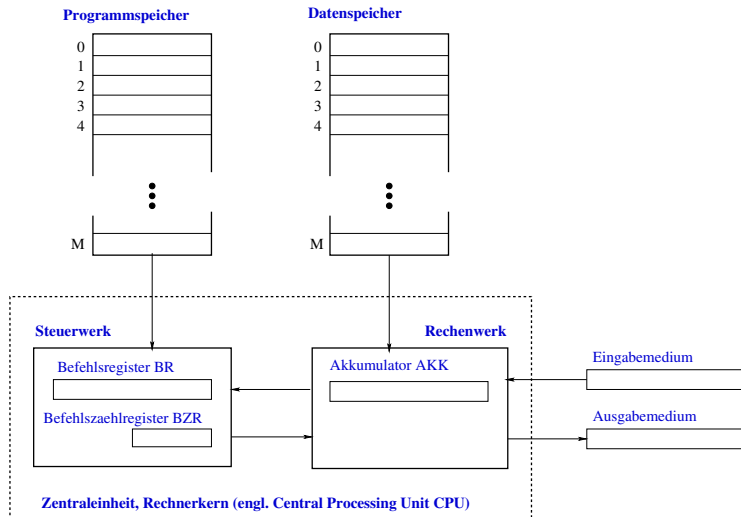
Kap. 7

Kap. 8

Kap. 9

# Der von-Neumann-(Programmier-) Stil

....geprägt durch die von-Neumann-Architektur von Rechnern:



# Evolution von Paradigmen und Sprachen (1)

...gekennzeichnet durch die **schrittweise Einführung** von **Abstraktionen** mit dem Ziel, **Einzelheiten** der zugrundeliegenden Rechenmaschine und **Programmausführung** immer mehr zu verbergen:

- ▶ **Assembler-Sprachen** führen mnemo-technische Instruktionsbezeichner und symbolische Marken ein, um **Maschinenbefehle** und **Programm- und Datenspeicheradressen** zu verbergen.
- ▶ **FORTRAN** führt Felder (engl. arrays) und Ausdrücke in mathematisch-üblicher Schreibweise ein, um **Register** zu verbergen.
- ▶ **ALGOL-ähnliche Sprachen** führen strukturierte Kontrollanweisungen ein, um **Sprungbefehle** und **Sprungmarken** zu verbergen (*'goto considered harmful'*).

# Evolution von Paradigmen und Sprachen (2)

- ▶ **Objektorientierte Sprachen** führen Sichtbarkeitssebenen und Kapselungen ein, um die **Datendarstellung und Speicherverwaltung** zu verbergen.
- ▶ **Deklarative Sprachen**, am bekanntesten **funktionale und logische Sprachen**, verbergen die **Auswertungsreihenfolge** und **verzichten dafür auf Kontrollanweisungen**. Reine deklarative Sprachen **verzichten** auch auf **Zuweisungen**, um **Seiteneffekte auszuschließen**.

In **deklarativen Sprachen** verschiebt sich dadurch die **Programmieraufgabe** von der

- ▶ **Festlegung der Rechenschritte zur Strukturierung der Anwendungsdaten und Beziehungen der Programmbestandteile.**

**Deklarative Sprachen** ähneln hierin formalen Spezifikations-sprachen, sind aber **ausführbar**.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Abgrenzung funktionaler u. logischer Sprachen

## Funktionale Sprachen

- ▶ beruhen auf dem **mathematischen Funktionsbegriff**.
- ▶ **Programme** sind **Systeme von Funktionen**, die über Gleichungen, Fallunterscheidungen und Rekursion definiert sind und auf (strukturierten) Daten arbeiten.
- ▶ bieten **effiziente, anforderungsgetriebene Auswertungsstrategien**, die auch die Arbeit mit (potentiell) unendlichen Strukturen unterstützen.

## Logische Sprachen

- ▶ beruhen auf **Prädikatenlogik**.
- ▶ **Programme** sind **Systeme von Prädikaten**, die durch eingeschränkte Formen logischer Formeln, z.B. Horn-Formeln (Implikation), definiert sind.
- ▶ bieten **Nichtdeterminismus** und **Prädikate mit mehreren Eingabe-/Ausgabemodi** zur Wiederverwendung von Code.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9



# Ziel all dieser Abstraktionen

...maßgebliche Beiträge zur Überwindung der sog. **Softwarekrise** zu leisten hin zu einer

- ▶ **ingenieurmäßigen Software-Entwicklung**  
(‘in time, in functionality, in budget’)
- ▶ **verifiziert, wartbar, erweiterbar**, etc.

indem dem Programmierer ein

- ▶ **angemessen(er)es** Abstraktionsniveau zur Formulierung, Modellierung und Lösung von Problemen

zur Verfügung gestellt wird.

# Prozedural vs. funktional: Ein Vergleich

Gegeben eine Aufgabe  $A$ , gesucht eine Lösung  $L$  für  $A$ .

**Prozedural:** Lösungsablauf typischerweise in 2 Schritten:

1. Ersinne ein algorithmisches Verfahren  $V$  zur Berechnung der Lösung  $L$  von  $A$ .
2. Codiere  $V$  als Folge von Anweisungen (Befehlen, Instruktionen) für den Rechner.

**Beachte:**

- **Schritt 2** erfordert hier zwingend, den Speicher explizit **anzusprechen** und zu **verwalten** (Allokation, Manipulation, Deallokation von Speicherzellen für Daten).

# Zur Illustration ein einfaches Beispiel (1)

*Aufgabe:* Liefere die Werte aller Einträge eines ganzzahligen Feldes mit einem Wert *von höchstens 10*.

Hier eine typische **prozedurale** Lösung, hier in **Pascal** (Argument in **Feldvariable a**, **Resultat** in **Feldvariable b**):

```
PROGRAM filter (input,output);
VAR a, b: ARRAY [1..maxLength] OF integer;
BEGIN
    (* Code zur Initialisierung von Feldvariable a *)
    FOR i:=1 TO maxLength DO a[i] := <Init.-Ausdruck>
    (* Hauptcode *)
    j := 1;
    FOR i:=1 TO maxLength DO
        IF a[i] <= 10 THEN BEGIN b[j] := a[i]; j := j+1 END
    END.
```

**Beachte:** Der Speicher wird explizit adressiert und manipuliert. Zusätzlich ist **'overhead' Code** erforderlich.

# Zur Illustration ein einfaches Beispiel (2)

...zum Vergleich hier eine typische funktionale Lösung, hier in Haskell:

```
a = [2,5..21]      ( [2,5..21] = [2,5,8,11,14,17,20] )  
b = [n | n <- a, n <= 10]
```

**Beachte:** Keine Speicheradressierung, -manipulation oder -verwaltung zur Berechnung von `b` erforderlich: `b ->> [2,5,8]`.  
Kein 'overhead' Code. Sogar noch knapper möglich:

```
b = [n | n <- [2,5..21], n <= 10]
```

Vergleiche die funktionale und mathematische Beschreibung

- ▶ `[ n | n <- a , n <= 10 ]`
- ▶  $\{ n \mid n \in a \wedge n \leq 10 \}$

unter dem Anspruch funktionaler Programmierung

- ▶ "...etwas von der *Eleganz der Mathematik* in die *Programmierung* zu bringen!"

# Essenz deklarativer Programmierung

...speziell auch **funktionaler** Programmierung: Betonung des

► **'was'** anstelle des **'wie'**!

**Deklarativ** (fkt.): Beschreibe, **was** wir bekommen möchten:

```
b = [n | n <- [2,5..21], n <= 10]
```

**Präskriptiv** (proz.): Beschreibe, **wie** wir etwas bek. möchten:

```
VAR a, b: ARRAY [1..maxLength] OF integer;  
(* Code zur Initialisierung von Feldvariable a *)  
FOR i:=1 TO maxLength DO a[i] := <Init.-Ausdruck>  
(* Hauptcode *)  
j := 1;  
FOR i:=1 TO maxLength DO  
  IF a[i] <= 10 THEN BEGIN b[j] := a[i]; j := j+1 END
```

Man soll den Menschen nie sagen,  
wie sie etwas tun sollen,  
sondern nur, was sie tun sollen.  
Dann wird ihr Einfallsreichtum einen verblüffen.

George S. Patton (1885-1945)  
amerik. General

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Automatische Listengenerierung

...mittels **Listenkomprehension** (engl. **list comprehension**) wie im Ausdruck

►  $[n \mid n \leftarrow a, n \leq 10]$  (vgl.  $\{n \mid n \in a \wedge n \leq 10\}$ )

...ist hierfür ein wichtiges, nützliches und für **funktionale Programmiersprachen** typisches **sprachliches Konstrukt** ohne Parallelen in nichtfunktionalen Sprachen.

**Anm.:** **Komprehension** – Zusammenfassung, Vereinigung von Mannigfaltigem zu einer Einheit (Philos.).

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Noch nicht überzeugt? Quicksort, Hoare-Stil

Betrachte eine komplexere Aufgabe: **Sortieren** à la **Hoare**.

**Aufgabe:** *Sortiere eine Liste  $L$  ganzer Zahlen aufsteigend.*

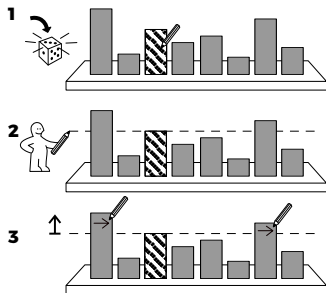
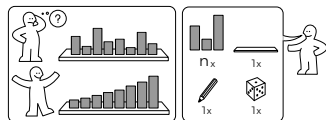
**Lösungsverfahren:** Das 'Teile und herrsche'-Sortierv Verfahren  
**Quicksort** von **Sir Tony Hoare (1961)**:

- ▶ **Teile:** Wähle ein Element  $I$  aus  $L$  und partitioniere  $L$  in zwei (möglicherweise leere) Teillisten  $L_1$  und  $L_2$ , so dass alle Elemente von  $L_1$  ( $L_2$ ) kleiner oder gleich (größer)  $I$  sind.
- ▶ **Herrsche:** Sortiere  $L_1$  und  $L_2$  mittels des **Quicksort**-Verfahrens (d.h. mittels rekursiver Aufrufe von **Quicksort**).
- ▶ **Kombiniere:** Bestimme Gesamtsortierung durch Zusammenführen der Teilsortierungen (hier trivial: konkateniere die sortierten Teillisten zur sortierten Gesamtliste).

# Quicksort, ~~IK~~DEA-Stil

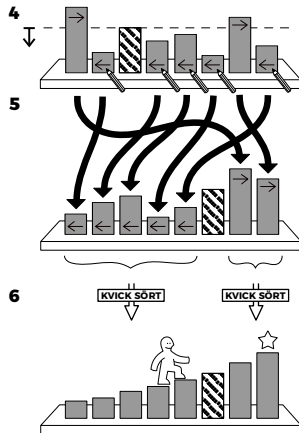
...Sortieren à la Hoare, die Grundidee so einfach, dass sie auch ~~IK~~DEA-erklärungsstilgeeignet ist (Feteke et al., TU Braunschw.):

## KVICK SÖRT



idea-instructions.com/quick-sort/  
v1.1, CC by-nc-sa 4.0

IDEA





# Quicksort, prozedural

...eine typische **prozedurale** Realisierung, hier in **Pseudocode**:

```
quickSort (L,low,high)
  if low < high
    then splitInd = partition (L,low,high)
         quickSort (L,low,splitInd-1)
         quickSort (L,splitInd+1,high) fi

partition (L,low,high)
  l = L[low]
  left = low
  for i = low+1 to high do
    if L[i] <= l then left = left+1
                                swap (L[i],L[left]) fi od
  swap (L[low],L[left])
  return left
```

**Aufruf:** quickSort(L,1,length(L)), wobei L die zu sortierende Liste ist, z.B. L=[4,2,3,4,1,9,3,3].

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Quicksort, funktional

...zum Vergleich eine typische funktionale Realisierung, hier in Haskell, mehr als eine Spur näher am Hoare- und IKDEA-Stil:

```
quickSort :: [Integer] -> [Integer]
quickSort [] = []
quickSort (n:ns) = quickSort [m | m <- ns, m <= n]
                  ++ [n]
                  ++ quickSort [m | m <- ns, m > n]
```

Aufrufe:

```
quickSort [] ->> []
quickSort [4,1,7,3,9] ->> [1,3,4,7,9]
quickSort [4,2,3,4,1,9,3,3] ->> [1,2,3,3,3,4,4,9]
```

# Übungsaufgabe 1.2.1.1

Betrachten Sie das ‘[Sieb des Eratosthenes](#)’ aus [Kapitel 1.1.1](#) als weiteres Beispiel. Realisieren Sie seine Verfahrensidee zur Berechnung von Primzahlen in

- ▶ [imperativen](#)

- ▶ prozeduralen (z.B. C, Pascal, Modula,...)
- ▶ objektorientierten (z.B. Java, C++, C#,...)

- ▶ [deklarativen](#)

- ▶ logischen (z.B. Prolog)

Programmiersprachen ihrer Wahl und vergleichen Sie Ihre Implementierungen mit der [Haskell](#)-Implementierung aus [Kapitel 1.1.1](#). Lässt sich die Verfahrensidee von Eratosthenes in manchen Sprachen einfacher umsetzen als in anderen? Kommt sie in allen Sprachen gleich gut und deutlich zum Ausdruck oder wird sie in manchen Sprachen von (Ausführungs-) Details überlagert?

# Kapitel 1.2.2

## Imperative vs. funktionale Programmierung

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

**1.2.2**

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Imperative Programmierung

...gekennzeichnet durch:

- ▶ Unterscheidung von **Ausdrücken** und **Anweisungen**.
- ▶ **Ausdrücke** liefern **Werte**; **Anweisungen** bewirken **Zustandsänderungen** (**Seiteneffekte**).
- ▶ **Programmausführung** ist die **Abarbeitung** von Anweisungen; dabei müssen auch **Ausdrücke** ausgewertet werden.
- ▶ **Kontrollflussspezifikation** mittels spezieller Anweisungen (Sequentielle Komposition, Fallunterscheidung, Schleifen, etc.)
- ▶ **Variablen** sind Verweise auf Speicherplätze. Ihre Werte können im Verlauf der Programmausführung (beliebig oft) geändert werden.
- ▶ Die **bewirkte Zustandsänderung** ist die **Bedeutung des Programms**.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Funktionale Programmierung

...gekennzeichnet durch:

- ▶ Ausschließlich **Ausdrücke**, keine **Anweisungen**.
- ▶ **Ausdrücke** liefern **Werte**. Zustandsänderungen gibt es nicht (und deshalb auch keine Seiteneffekte).
- ▶ **Programmausführung** ist **Auswertung** eines **Ausdrucks**. Sein **Wert** ist Ergebnis und **Bedeutung des Programms**.
- ▶ Keine Kontrollflussspezifikation; allein **Datenabhängigkeiten** steuern die Auswertung(sreihenfolge).
- ▶ **Variablen** (besser: **Namen**) sind an **Ausdrücke** gebunden. Einmal ausgewertet, ist eine Variable (Name) an einen einzelnen Wert gebunden; ein späteres Überschreiben oder Neubelegen ist nicht möglich.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Imperative Programmierung

...befehlsbasierte Programmierung:

- ▶ **Programm:** Menge von Befehlen (oder Instruktionen oder Anweisungen) strukturiert durch ein Regelwerk (in Form von Kontrollflussanweisungen (*if-then-else*, *while-do*, *for-do*,...)), das ihre Ausführungsabfolge festlegt und steuert.
- ▶ **Bedeutung von Befehlen und Programmen:** Zustandsänderungen (Zustand: Zuordnung von Werten an Variablen ' $\sigma : \text{Variablen} \rightarrow \text{Werte}$ '. Zustandsänderung: Neuordnung von Werten an Variablen; Bsp.: Angesetzt auf  $\sigma$  ordnet der Befehl  $x := x + y$  Variable  $x$  die Summe aus  $\sigma(x)$  und  $\sigma(y)$  neu als Wert zu und lässt alle anderen Zuordnungen unverändert).
- ▶ **Vorlegbare Frage an ein Programm:** Was sind die Werte der Variablen nach Terminierung von einem Programm  $\pi_{imp}$ , d.h. in welchem (End-) Zustand  $\sigma'$  terminiert  $\pi_{imp}$  angesetzt auf einen (Anfangs-) Zustand  $\sigma$ ?

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Funktionale Programmierung

...gleichungsbasierte Programmierung:

- ▶ **Programm:** Menge von **Vereinbarungen** von **Wertegleichheiten** von **Ausdrücken** (`pi = 3.14`, `quadrat n = n*n`, `isodd n = iseven (n-1)`, `binom m n = ...`).
- ▶ **Bedeutung von Ausdrücken und Programmen:** Von **Ausdrücken**, ihr **Wert** (Bsp.: Der Wert von Ausdruck `binom 45 6` ist die natürliche Zahl `8.145.060`, von Ausdruck `quadrat 8` die natürliche Zahl `64`); von **Programmen**, der **Wert** ihres 'Hauptausdrucks' (`main = ...`).
- ▶ **Vorlegbare Frage an ein Programm:** Was ist der **Wert** von einem **Ausdruck** `ausd`, wenn `ausd` unter Rückgriff auf die im Programm  $\pi_{fkt}$  definierten Ausdruckswertgleichheiten ausgewertet wird?

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9



# Kapitel 1.2.3

## Von imperativer zu funktionaler Programmierung

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

**1.2.3**

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Haupttherausforderung

...konzeptuell und praktisch den Übergang von

► befehlsorientierter



zu

► ergebnisorientierter



Denk- und Handlungsweise zu meistern!

Dabei gilt:

Omne initium difficile.  
Aller Anfang ist schwer.

lat., sprichwörtl.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Versprechen

...ist dieser **Übergang gemeistert**, sind auch das neue Paradigma **funktionaler Programmierung** und die Inhalte dieser **LVA** i.w. gemeistert.

**Tipp:** Fällt es anfangs schwer, weil **ungewohnt**, eine Aufgabe **funktional** zu lösen, so lohnt es sich, zu diesem Abschnitt zurückzukehren und zu überlegen, ob die Ursache der empfundenen Schwierigkeit möglicherweise darin liegt, die Aufgabe in **befehls-** statt **ergebnisorientierter Denk- und Handlungsweise** angehen zu wollen (s. auch **Anh. A**, insb. **A.6** bis **A.9**).

Der Mensch beginnt nicht leicht zu denken.  
Sobald er aber erst einmal den Anfang damit gemacht hat,  
hört er nicht mehr damit auf.

Jean-Jacques Rousseau (1712-1778)  
franz.-schweizer. Philosoph

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Übungsaufgabe 1.2.3.1

Macht euch nichts draus,  
ich weiß, ihr werdet das nie verstehen.

Ludwig Wittgenstein (1889-1951)

österreich. Philosoph

Namenspatron für den höchstdotierten  
österreich. Wissenschaftspreis

Generalaufgabe für dieses (und alle folgenden) Semester:

Widerleg(t) Wittgenstein!

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 1.2.4

## Funktionale Programmierung: Stärken, Schwächen

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

**1.2.4**

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Stärken und Vorteile fkt. Programmierung

## ► Einfach(er) zu erlernen

...da wenige(r) Grundkonzepte (vor allem keinerlei (Ma-  
schinen-) Instruktionen; deshalb auch keine Zuweisungen,  
keine Schleifen, keine Sprünge)

## ► Höhere Produktivität

...da Programme signifikant kürzer als funktional ver-  
gleichbare imperative Programme sind (Faktor 5 bis 10)

## ► Höhere Zuverlässigkeit

...da Korrektheitsüberlegungen und -beweise einfach(er)  
sind (math. Fundierung, keine durchscheinende Maschine)

# Illustration auf der syntaktischen Ebene

## Konzeptuell:

- Die unendliche Folge der natürlichen Zahlen ab 1:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...

Mögliche Kurzschreibweisen:

- 1. (zu knapp, Verwechslungsgefahr mit **1-tens**)
- 1... (zu lang)
- 1.., 1ff (*fortfolgende*),  $1\phi$ ,  $\mathbb{IN}_1$ ,  $\mathbb{IN}$ , ... (genau richtig)

In Haskell:

`[1..]` (Synt. Zucker: Eckige Klammern zur Listenkennzeichnung)

In Java:

```
class Naturals implements Iterator<Integer> {  
    private int value = 1;  
    public boolean hasNext() { return true; }  
    public Integer next() { return value++; }  
}
```

(deutlich mehr syntakt. Zucker)

# Schwächen und Nachteile fkt. Programmierung

## ► Geringe(re) Performanz

**Aber:** Große Fortschritte sind erzielt (Performanz oft vergleichbar mit entsprechenden C-Implementierungen); Korrektheit zudem vorrangig gegenüber Performanz; einfache(re) Parallelisierbarkeit fkt. Programme.

## ► Manchmal unangemessen, oft für inhärent zustandsbasierte Anwendungen, zur GUI-Programmierung

**Aber:** Eignung einer Methode/Technologie/**Programmierstils** für einen Anwendungsfall ist stets zu untersuchen und überprüfen; dies ist kein Spezifikum fkt. Programmierung.

**Außerdem:** **Unterstützung zustandsbehafteter Programmierung** in vielen funktionalen Programmiersprachen durch spezielle Mechanismen; z.B. in **Haskell** durch das **Monadenkonzept** (s. LVA 185.A05 Fortgeschrittene funktionale Programmierung).

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9



Es ist schwieriger, eine vorgefasste Meinung  
zu zertrümmern als ein Atom.

Albert Einstein (1879-1955)  
dt. schweiz.-amerik. Physiker

...häufig vorgebrachte Schwächen und Nachteile funktionaler  
Programmierung insgesamt (oft) nur **vorurteilsbehaftet** und  
**vermeintlich**.

Lass dich nicht durch Vorurteile bestimmen [...].

1. Timotheus 5,21

# Einsatzfelder funktionaler Programmierung

...mittlerweile ‘überall’:

- ▶ Curt J. Simpson. [Experience Report: Haskell in the “Real World”](#): Writing a Commercial Application in a Lazy Functional Language. In Proc. 14th ACM SIGPLAN Int. Conf. on Funct. Prog. (ICFP 2009), 185-190, 2009.
- ▶ Jerzy Karczmarczuk. [Scientific Computation and Functional Programming](#). Computing in Science and Engineering 1(3):64-72, 1999.
- ▶ Bryan O’Sullivan, John Goerzen, Don Stewart. [Real World Haskell](#). O’Reilly, 2008.
- ▶ Yaron Minsky. [OCaml for the Masses](#). Communications of the ACM, 54(11):53-58, 2011.
- ▶ Michael Snoyman. [Developing Web Applications with Haskell and Yesod](#). O’Reilly, 2012.
- ▶ [Haskell in Industry and Open Source](#):  
[www.haskell.org/haskellwiki/Haskell\\_in\\_industry](http://www.haskell.org/haskellwiki/Haskell_in_industry)

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

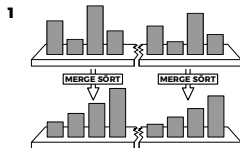
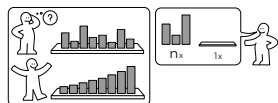
Kap. 8

Kap. 9

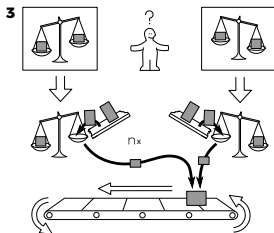
# Übungsaufgabe 1.2.4.1: Aufwand, Performanz

Implementieren Sie die Verfahren *Kvick Sört* und *Merge Sört* zum Sortieren ganzzahliger Listen in Haskell und in einer weiteren Programmiersprache Ihrer Wahl. Vergleichen Sie die Implementierungen hinsichtlich Programmieraufwand und Laufzeitverhalten durch Laufzeitmessungen für Argumentlisten wachsender Länge.

## MERGE SÖRT



idea-instructions.com/merge-sort/  
v1.1, CC by-nc-sa 4.0



# Kapitel 1.2.5

## Warum funktionale Programmierung mit Haskell?

Facta loquuntur.  
Tatsachen sprechen (lassen).  
lat., sprichwörtl.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

**1.2.5**

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Funktionale Programmiersprachen

...vielfältig und **zahlreich**, z.B.:

- ▶  **$\lambda$ -Kalkül** (späte 1930er Jahre, Alonzo Church, Stephen Kleene)
- ▶ **Lisp** (frühe 1960er Jahre, John McCarthy)
- ▶ **ML, SML** (Mitte 1970er Jahre, Michael Gordon, Robin Milner)
- ▶ **Hope** (um 1980, Rod Burstall, David McQueen)
- ▶ **Miranda** (um 1980, David Turner)
- ▶ **OPAL** (Mitte 1980er Jahre, Peter Pepper et al.)
- ▶ **Haskell** (späte 1980er Jahre, Paul Hudak, Philip Wadler et al.)
- ▶ **Gofer** (frühe 1990er Jahre, Mark Jones)
- ▶ ...

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Warum also gerade Haskell?

Learning Haskell opens one's mind to  
new programming paradigms, which  
might produce clearer and shorter  
implementations.

Mihai Maruseac

*Haskell: A Language for Modern Times.*  
Crossroads, the ACM Magazine  
for Students 24(1):64-66, 2017.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Warum also nicht Haskell?

...Haskell ist eine

## ► fortgeschrittene moderne funktionale Sprache

- starke Typisierung
- verzögerte Auswertung (lazy evaluation)
- Funktionen höherer Ordnung/Funktionale
- Polymorphie/Generizität
- Musterpassung (pattern matching)
- Datenabstraktion (abstrakte Datentypen)
- Modularisierung (für Programmierung im Großen)
- ...

## ► Sprache für 'realistische (real world)' Probleme

- mächtige Bibliotheken
- Schnittstellen zu anderen Sprachen, z.B. zu C
- ...

In Summe: **Haskell** ist reich; zugleich ist es eine **gute** Lehrsprache; auch dank des Interpretierers **Hugs**! Und **Haskell** ist mehr als das!

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9





# RedMonk Programming Language Rankings

Rg	Jan.'15	Rg	Jun'15	Rg	Jan'16	Rg	Jun'16	Rg	Jan'17	Rg	Jun'17
1	JS	1	JS	1	JS	1	JS	1	JS	1	JS
2	Java	2	Java	2	Java	2	Java	2	Java	2	Java
3	PHP	3	PHP	3	PHP	3	PHP	3	Python	3	PHP
4	Python	4	Python	4	Python	4	Python	4	PHP	4	Python
5	C#	5	C#	5	C#	5	C#	5	C#	5	C#
6	C++	5	C++	5	C++	5	C++	5	C++	5	C++
7	Ruby	5	Ruby	5	Ruby	5	Ruby	7	CSS	5	Ruby
8	CSS	8	CSS	8	CSS	8	CSS	7	Ruby	8	CSS
9	C	9	C	9	C	9	C	9	C	9	C
10	Obj-C	10	Obj-C	10	Obj-C	10	Obj-C	10	Obj-C	10	Obj-C
11	Perl	11	Perl	11	Shell	11	Shell	11	Scala	11	Shell
12	Shell	11	Shell	12	Perl	12	R	11	Shell	12	R
13	R	13	R	13	R	13	Perl	11	Swift	13	Perl
14	Scala	14	Scala	14	Scala	14	Scala	14	R	14	Scala
15	Haskell	15	Go	15	Go	15	Go	15	Go	15	Go
16	Matlab	15	Haskell	15	Haskell	16	Haskell	15	Perl	16	Haskell
17	Go	17	Matlab	17	Swift	17	Swift	17	TS	17	Swift
18	VB	18	Swift	18	Matlab	18	Matlab	18	PS	18	Matlab
19	Clojure	19	Groovy	19	Clojure	19	VB	19	Haskell	19	VB
20	Groovy	19	VB	19	Groovy	20	Clojure	20	Clojure	20	Clojure
					VB	20	Groovy	20	CS		20 Groovy
								20	Lua		
								20	Matlab		

## Abkürzungen:

JS	JavaScript	TS	TypeScript
Obj-C	Objective-C	PS	PowerShell
VB	Visual Basic	CS	CoffeeScript

URL: <http://redmonk.com>

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Übungsaufgabe 1.2.5.1

Überprüfe, wie [Haskell](#) in den [RedMonk](#)-Ranglisten aus dem

- ▶ [Januar 2018](#), [Juni 2018](#)
- ▶ [Januar 2019](#), [Juni 2019](#)

abschneidet. Welche Verschiebungen hat es generell gegeben?  
Welche für [Haskell](#)?

# Steckbrief 'Funktionale Programmierung'

Grundlage:	Lambda- ( $\lambda$ -) Kalkül; Basis formaler Berechenbarkeitsmodelle
Abstraktionsprinzip:	Funktionen (höherer Ordnung)
Charakt. Eigenschaft:	Referentielle Transparenz
Historische und aktuelle Bedeutung:	Basis vieler Programmiersprachen; praktische Ausprägung auf dem $\lambda$ -Kalkül basierender Berechenbarkeitsmodelle
Anwendungsbereiche:	Theoretische Informatik, Künstliche Intelligenz (Expertensysteme), Experimentelle Software/Prototypen, Programmierunterricht,..., <b>Software-Lsg. industriellen Maßstabs</b>
Programmiersprachen:	Lisp, ML, Miranda, Haskell,...

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Steckbrief 'Haskell'

Benannt nach:	<b>Haskell B. Curry</b> (1900-1982) <a href="http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Curry.html">www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Curry.html</a>
Paradigma:	Rein funktionale Programmierung
Eigenschaften:	Lazy evaluation, pattern matching
Typsicherheit:	Stark typisiert, Typinferenz, modernes polymorphes Typsystem
Syntax:	Komprimiert, kompakt, intuitiv
Informationen:	<a href="http://haskell.org">http://haskell.org</a> <a href="http://haskell.org/tutorial/">http://haskell.org/tutorial/</a>
Interpreter:	Hugs ( <a href="http://haskell.org/hugs/">haskell.org/hugs/</a> )
Compiler:	Glasgow Haskell Compiler (GHC)

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 1.2.6

## Erste Schritte in Haskell: Gewöhnliche, literate Haskell-Skripte

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

**1.2.6**

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Haskell-Programme

...gibt es in zwei sich konzeptuell und notationell unterscheidenden Varianten.

Als sog.

- ▶ (Gewöhnliches) Haskell-Skript

...alles, was nicht notationell als Kommentar ausgezeichnet ist, wird als Programmtext betrachtet.

Konvention: `.hs` als Dateiendung

- ▶ Literates Haskell-Skript (engl. `literate Haskell Script`)

...alles, was nicht notationell als Programmtext ausgezeichnet ist, wird als Kommentar betrachtet.

Konvention: `.lhs` als Dateiendung

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# myFirstScript.hs: Gewöhnliches Haskell-Skript

```
{- myFirstScript.hs: Gewöhnliche Skripte erhalten  
konventionsgemäß die Dateiendung .hs -}
```

```
-- Fakultätsfunktion
```

```
fac :: Integer -> Integer
```

```
fac n = if n == 0 then 1 else n * fac (n-1)
```

```
-- Binomialkoeffizienten
```

```
binom' :: (Integer,Integer) -> Integer
```

```
binom' (n,k) = div (fac n) (fac k * fac (n-k))
```

```
-- Konstante (0-stellige) Funktion sechs_aus_45
```

```
sechs_aus_45 :: Integer
```

```
sechs_aus_45 = (fac 45) 'div' (fac 6 * fac (45-6))
```

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# myFirstLitScript.lhs: Literates Haskell-Skript

myFirstLitScript.lhs: Literate Skripte erhalten  
konventionsgemäß die Dateierendung .lhs

## Fakultätsfunktion

```
> fac :: Integer -> Integer
> fac n = if n == 0 then 1 else n * fac(n-1)
```

## Binomialkoeffizienten

```
> binom' :: (Integer,Integer) -> Integer
> binom' (n,k) = div (fac n) (fac k * fac (n-k))
```

## Konstante (0-stellige) Funktion sechs\_aus\_45

```
> sechs_aus_45 :: Integer
> sechs_aus_45 = (fac 45) 'div' (fac 6 * fac (45-6))
```

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

120/170



# Kommentare in Haskell-Programmen

## Kommentare in

- ▶ (gewöhnlichem) Haskell-Skript
  - **Einzeilig**: Alles nach `--` bis zum Rest der Zeile
  - **Mehrzeilig**: Alles zwischen `{-` und `-}`
- ▶ **literatem** Haskell-Skript
  - Jede nicht durch `>` eingeleitete Zeile  
(**Beachte**: Kommentar- und Codezeilen müssen durch mindestens eine Leerzeile getrennt sein.)

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Das Haskell-Vokabular

## 21 Schlüsselwörter, mehr nicht:

```
case class data default deriving do else
if  import in  infix infixl infixr instance
let module newtype of then type where
```

## Schlüsselwörter haben

- ▶ wie in anderen Programmiersprachen eine besondere Bedeutung und dürfen nicht als **Identifikatoren** für z.B. Funktionen oder Funktionsargumente verwendet werden.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Identifikatoren

...sind:

- ▶ Nichtleere Zeichenfolgen aus Klein- und Großbuchstaben, Ziffern, einfachen Hochkommata ' und Unterstrichen \_, die mit einem Buchstaben beginnen.

Die Verwendung des Identifikators (als Funktionsname, Typname, Typvariable, etc.) legt fest, ob der erste Buchstabe ein

- ▶ Kleinbuchstabe (z.B. für Funktionsnamen, Typvariablen)
- ▶ Großbuchstabe (z.B. für Typnamen)

sein muss.

# Module Prelude: Standard-Präludium

## Die Definitionen

- ▶ einiger der in diesem Kapitel beispielhaft betrachteten Rechenvorschriften und vieler weiterer allgemein nützlicher Deklarationen von Typen und Rechenvorschriften finden sich im vordefinierten Modul [Prelude](#), dem sog. [Standard-Präludium](#) (engl. [Standard Prelude](#)).

## Das quelloffene [Standard-Präludium](#)

- ▶ wird [automatisch](#) mit jedem Haskell-Programm geladen, so dass die darin definierten Typen und Rechenvorschriften stets zur Verfügung stehen.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.2.1

1.2.2

1.2.3

1.2.4

1.2.5

1.2.6

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Tipp

Nachschlagen und lesen im **Standard-Präludium** ist

- ▶ gute und einfache Möglichkeit, sich mit der **Syntax** von **Haskell** vertraut zu machen und ein **Gefühl** für den **Stil funktionaler Programmierung** zu entwickeln.

‘Haskell 98’-Sprachbericht:

- ▶ Simon Peyton Jones (Hrsg.). **Haskell 98: Language and Libraries. The Revised Report**. Cambridge University Press, 2003. [www.haskell.org/definitions](http://www.haskell.org/definitions). (Kapitel 8, Standard Prelude; Kapitel 8.1, Module Prelude)
- ▶ **Standard-Präludium**.  
<http://www.haskell.org/onlinereport/standard-Prelude.html>

# Kapitel 1.3

Nützliche Werkzeuge für Haskell: Hugs,  
GHC, GHCi, Hoogle, Hayoo, Leksah

Inhalt

Teil I

Kap. 1

1.1

1.2

**1.3**

1.3.1

1.3.2

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Überblick

...beispielhaft 5 nützliche Werkzeuge für die funktionale Programmierung in Haskell:

1. **Hugs**: Ein Haskell-Interpreter
2. **GHC, GHCi**: Ein Haskell-Übersetzer, ein Haskell-Interpreter
3. **Hoogle, Hayoo**: Zwei Haskell(-spezifische) Suchmaschinen
4. **Leksah**: Eine (in Haskell geschriebene) quelloffene integrierte Entwicklungsumgebung IDE

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

127/170

# Kapitel 1.3.1

## Hugs

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

**1.3.1**

1.3.2

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV



# Hugs

...ein (weiterhin) populärer Haskell-Interpreter (mit allerdings eingestellter Entwicklung).

Hugs im Netz:

- ▶ [www.haskell.org/hugs](http://www.haskell.org/hugs)

Zur Arbeit mit Hugs siehe z.B.:

- ▶ H. Conrad Cunningham. *Notes on Functional Programming with Haskell*. Course Notes, University of Mississippi, 2007. [citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf) (Chapter 4, Using the Hugs Interpreter)

# Hugs-Aufruf ohne Programmskript

Aufruf von **Hugs** ohne Skript:

```
hugs
```

Nach Aufruf steht die **Taschenrechnerfunktionalität** von **Hugs** (sowie im Prelude definierte Funktionen) zur **Auswertung von Ausdrücken** zur Verfügung:

```
Main>47*100+11
```

```
4711
```

```
Main>reverse "stressed"
```

```
"desserts"
```

```
Main>length "desserts"
```

```
8
```

```
Main>(4>17) || (17+4==21)
```

```
True
```

```
Main>True && False
```

```
False
```

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Hugs-Aufruf mit Programmskript

Aufruf von **Hugs** mit Skript:

```
hugs <filename>
```

Zum Beispiel: `hugs myFirstScript.hs`  
`hugs myFirstScript.lhs`

Nach Aufruf stehen zusätzlich zu den Präludiumsfunktionen auch alle im geladenen Skript deklarierten Funktionen zur Verfügung:

```
Main>fac 6
720
Main>binom' (49,6)
13.983.816
Main>sechs_aus_45
8.145.060
```

Das **Hugs**-Kommando `:l(oad)` erlaubt ein anderes Skript zu laden (wodurch ein eventuell vorher geladenes Skript ersetzt wird):

```
Main>:l myFirstScript.lhs
```

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Wichtige Hugs-Kommandos

<code>:?</code>	Liefert Liste der <b>Hugs</b> -Kommandos
<code>:load &lt;fileName&gt;</code>	Lädt die Haskell-Datei <fileName> (erkennbar an Endung <code>.hs</code> bzw. <code>.lhs</code> )
<code>:reload</code>	Wiederholt letztes Ladekommando
<code>:quit</code>	Beendet den aktuellen <b>Hugs</b> -Lauf
<code>:info name</code>	Liefert Information über das mit <code>name</code> bezeichnete 'Objekt'
<code>:type exp</code>	Liefert den Typ des Argumentausdrucks <code>exp</code>
<code>:edit &lt;fileName&gt;.hs</code>	Öffnet die Datei <fileName>.hs enthaltende Datei im voreingestellten Editor
<code>:find name</code>	Öffnet die Deklaration von <code>name</code> im voreingestellten Editor
<code>!&lt;com&gt;</code>	Ausführen des Unix- oder DOS-Kommandos <com>

Alle Kommandos können mit dem ersten Buchstaben abgekürzt werden.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

132/170

# Hugs-Fehlermeldungen und -warnungen

## ► Fehlermeldungen

### ► Syntaxfehler

```
Main> sechsAus45 == 123456) ...liefert  
ERROR: Syntax error in input (unexpected '')
```

### ► Typfehler

```
Main> sechsAus45 + False ...liefert  
ERROR: Bool is not an instance of class "Num"
```

### ► Programmfehler

...später

### ► Modulfehler

...später

## ► Warnungen

### ► Systemmeldungen

...später

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

133/170

# Bibliotheken: Professionell und praxisgerecht

- ▶ **Haskell** stellt umfangreiche **Bibliotheken** mit vielen vordefinierten Funktionen zur Verfügung.
- ▶ Das sog. **Standard-Präludium** (engl. **Standard Prelude**) wird automatisch beim Start von **Hugs** geladen und stellt eine Vielzahl von Funktionen bereit, z.B. zum
  - ▶ Umkehren von Zeichenreihen bzw. genereller, allgemein von Listen beliebiger Typen (**reverse**)
  - ▶ Ver- und entpaaren von Listen (**zip**, **unzip**)
  - ▶ Aufsummieren und Aufmultiplizieren von Elementen einer numerischen Liste (**sum**, **product**)
  - ▶ ...

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

134/170

# Namenskonflikte mit vordefinierten Namen

...soll eine Funktion eines gleichen (bereits in `Prelude.hs` vordefinierten) Namens deklariert werden, können Namenskonflikte durch `verstecken` (engl. `hiding`) vordefinierter Namen vermieden werden.

Am Beispiel von `reverse`, `zip`, `sum`:

- Füge die Zeile

```
import Prelude hiding (reverse,zip,sum)
```

am Anfang des Haskell-Skripts im Anschluss an die Modul-Anweisung (so vorhanden) ein; dadurch werden die vordefinierten Namen `reverse`, `zip` und `sum` verborgen.

(Mehr dazu in [Kapitel 17](#) im Zusammenhang mit dem [Modulkonzept](#) von Haskell).

# Kapitel 1.3.2

## GHC, GHCi

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

**1.3.2**

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV



# GHC, GHCi

...ein populärer Haskell-Compiler:

- ▶ Glasgow Haskell Compiler (GHC)

...und ein damit verbundener Interpretierer:

- ▶ GHCi (GHC interactive)

GHC und GHCi im Netz:

- ▶ [hackage.haskell.org/platform](http://hackage.haskell.org/platform)

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Kapitel 1.3.3

## Hoogle, Hayoo

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

**1.3.3**

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Hoogle, Hayoo

...zwei nützliche [Suchmaschinen](#), um vordefinierte Funktionen (in Haskell-Bibliotheken) aufzuspüren.

[Hoogle](#) und [Hayoo](#) unterstützen die Suche nach

- ▶ Funktionsnamen
- ▶ Modulnamen
- ▶ Funktionssignaturen

[Hoogle](#) und [Hayoo](#) im Netz:

- ▶ [www.haskell.org/hoogle](http://www.haskell.org/hoogle)
- ▶ [hayoo.fh-wedel.de](http://hayoo.fh-wedel.de)

# Kapitel 1.3.4

## Leksah

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

**1.3.4**

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Leksah

...eine **quelloffene in Haskell geschriebene IDE** mit GTK-Oberfläche für Linux, Windows und MacOS.

## Unterstützte Eigenschaften:

- ▶ **Quell-Editor** zur Quellprogrammerstellung.
- ▶ **Arbeitsbereiche** zur Verwaltung von Haskell-Projekten in Form eines oder mehrerer Cabal-Projekte.
- ▶ **Cabal-Paketverwaltung** zur Verwaltung von Versionen, Übersetzeroptionen, Testfällen, Haskell-Erweiterungen, etc.
- ▶ **Modulbetrachter** zur Projektinspektion.
- ▶ **Debugger** auf Basis eines integrierten ghc-Interpreterers.
- ▶ **Erweiterte Editorfunktionen** mit Autovervollständigung, 'Spring-zu-Fehlerstelle'-Funktionalität, etc.
- ▶ ...

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.3.4

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Leksah (fgs.)

Leksah im Netz:

- ▶ [www.leksah.org](http://www.leksah.org)

Anmerkung:

- ▶ Teils aufwändige Installation, oft vertrackte Versionsabhängigkeiten zwischen Komponenten.
- ▶ Für die Zwecke der LVA nicht benötigt.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

**1.3.4**

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Kapitel 1.4

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

**1.4**

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9





Teil IV

Kap. 10

Kap. 11

143/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (1)

-  Christopher Allen, Julie Moronuki. *Haskell Programming from First Principles*. ebook. <http://haskellbook.com>
-  Sergio Antoy, Michael Hanus. *Functional Logic Programming*. Communications of the ACM 53(4):74-85, 2010.
-  John W. Backus. *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*. Communications of the ACM 21(8):613-641, 1978.
-  Henri E. Baal, Dick Grune. *Programming Language Essentials*. Addison-Wesley, 1994. (Chapter 4, Functional Languages; Chapter 7, Other Paradigms)

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV




Kap. 10

Kap. 11

-144/170



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (2)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 1, Motivation und Einführung)
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 1, What is functional programming? Kapitel 2.1, A session with GHCi)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 1, Einführung; Kapitel 2, Programmierumgebung; Kapitel 4.1, Rekursion über Zahlen; Kapitel 6, Die Unix-Programmierumgebung)

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

145/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (3)



H. Conrad Cunningham. *Notes on Functional Programming with Haskell*. Course Notes, University of Mississippi, 2007. [citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf](https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf) (Chapter 1.2, Excerpts from Backus' 1977 Turing Award Address; Chapter 1.3, Programming Language Paradigms; Chapter 1.4, Reasons for Studying Functional Programming; Chapter 1.5, Objections Raised Against Functional Programming; Chapter 4, Using the Hugs Interpreter)



Hal Daumé III. *Yet Another Haskell Tutorial*. [wikibooks.org](https://en.wikibooks.org)-Ausgabe, 2007.  
[https://en.wikibooks.org/wiki/Yet\\_Another\\_Haskell\\_Tutorial](https://en.wikibooks.org/wiki/Yet_Another_Haskell_Tutorial)

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

146/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (4)



Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 1.1, The von Neumann Bottleneck; Kapitel 1.2, Von Neumann Languages)



Frank DeRemer, Hans H. Kron. *Programming-in-the-Large vs. Programming-in-the-Small*. IEEE Transactions on Software Engineering 2(2):80-86, 1976.



Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *The Architecture of the Utrecht Haskell Compiler*. In Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009), 93-104, 2009.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-147/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (5)

-  Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *UHC Utrecht Haskell Compiler*, 2009. [www.cs.uu.nl/wiki/UHC](http://www.cs.uu.nl/wiki/UHC)
-  Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 1, Erste Schritte; Anhang A, Zur Benutzung des Systems)
-  Chris Done. *Try Haskell*. Online Hands-on Haskell Tutorial. [tryhaskell.org](http://tryhaskell.org).
-  Robert W. Floyd. *The Paradigms of Programming*. Turing Award Lecture. Communications of the ACM 22(8):455-460, 1979.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9






Teil IV

Kap. 10

Kap. 11

148/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (6)

-  Neal Ford. *Functional Thinking: Why Functional Programming is on the Rise*. IBM developerWorks, 11 pages, 2013. [www.ibm.com/developerworks/java/library/j-ft20/j-ft20-pdf.pdf](http://www.ibm.com/developerworks/java/library/j-ft20/j-ft20-pdf.pdf)
-  Bastiaan Heeren, Daan Leijen, Arjan van IJzendoorn. *Helium, for Learning Haskell*. In Proceedings of the ACM SIGPLAN 2003 Haskell Workshop (Haskell 2003), 62-71, 2003.
-  Konrad Hinsen. *The Promises of Functional Programming*. Computing in Science and Engineering 11(4):86-90, 2009.
-  C.A.R. Hoare. *Algorithm 64: Quicksort*. Communications of the ACM 4(7):321, 1961.
-  C.A.R. Hoare. *Quicksort*. The Computer Journal 5(1):10-15, 1962.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9




Teil IV

Kap. 10

Kap. 11

-149/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (7)

-  Paul Hudak. *Conception, Evolution and Applications of Functional Programming Languages*. Communications of the ACM 21(3):359-411, 1989.
-  Paul Hudak, Joseph Fasel, John Peterson. *A Gentle Introduction to Haskell*. Technischer Bericht, Yale University, 1996. <https://www.haskell.org/tutorial>
-  John Hughes. *Why Functional Programming Matters*. The Computer Journal 32(2):98-107, 1989.
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 1, Introduction; Kapitel 2, First Steps)
-  Arjan van IJzendoorn, Daan Leijen, Bastiaan Heeren. *The Helium Compiler*. [www.cs.uu.nl/helium](http://www.cs.uu.nl/helium).

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9






Teil IV

Kap. 10

Kap. 11

150/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (8)

-  Mark P. Jones, Alastair Reid et al. *The Hugs98 User Manual*, 1999. [www.haskell.org/hugs](http://www.haskell.org/hugs)
-  Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. Computing in Science and Engineering 1(3):64-72, 1999.
-  Donald E. Knuth. *Literate Programming*. The Computer Journal 27(2):97-111, 1984.
-  Konstantin Läufer, George K. Thiruvathukal. *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*. Computing in Science and Engineering 11(5):68-75, 2009.
-  Mihai Maruseac. *Haskell: A Language for Modern Times*. Crossroads, the ACM Magazine for Students 24(1):64-66, 2017.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-151/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (9)



Martin Odersky. *Funktionale Programmierung*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 599-612, 2006. (Kapitel 5.1, Funktionale Programmiersprachen; Kapitel 5.2, Grundzüge des funktionalen Programmierens)



Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 1, Getting Started)



Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 1, Was die Mathematik uns bietet; Kapitel 2, Funktionen als Programmiersprache)

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV




Kap. 10

Kap. 11

152/170



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (10)

-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006. (Kapitel 1, Grundlagen der funktionalen Programmierung)
-  Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009. (Chapter 1, Thinking differently; Chapter 2, Core concepts in functional programming)
-  Simon Peyton Jones (Hrsg.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. [www.haskell.org/definitions](http://www.haskell.org/definitions). (Kapitel 8, Standard Prelude; Kapitel 8.1, Module Prelude)

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-153/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (11)



Chris Sadler, Susan Eisenbach. *Why Functional Programming?* In *Functional Programming: Languages, Tools and Architectures*, Susan Eisenbach (Hrsg.), Ellis Horwood, 9-20, 1987.



Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.



Curt J. Simpson. *Experience Report: Haskell in the “Real World”*: Writing a Commercial Application in a Lazy Functional Language. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8




Kap. 9

Teil IV

Kap. 10

Kap. 11

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (12)

-  Simon Thompson. *Where Do I Begin? A Problem Solving Approach in Teaching Functional Programming*. In Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97), Springer-Verlag, LNCS 1292, 323-334, 1997.
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999.  
(Kapitel 1, Introducing functional programming; Kapitel 2, Getting started with Haskell and Hugs)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.  
(Kapitel 1, Introducing functional programming; Kapitel 2, Getting started with Haskell and GHCi)

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-155/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (13)

-  Reinhard Wilhelm, Helmut Seidl. *Compiler Design – Virtual Machines*. Springer-V., 2010. (Kapitel 3, Functional Programming Languages; Kapitel 3.1, Basic Concepts and Introductory Examples)
-  Hugs-Benutzerhandbuch. *The Hugs98 User Manual*.  
<https://www.haskell.org/hugs/pages/hugsman/index.html>
-  GHCi-Benutzerhandbuch. *Glasgow Haskell Compiler User's Guide*. [http://www.haskell.org/ghc/docs/latest/html/users\\_guide/ghci.html](http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci.html)
-  Haskell's Standard-Präludium.  
<https://www.haskell.org/onlinereport/standard-prelude.html>

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV





Kap. 10

Kap. 11

-156/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 1 (14)

Welches Paradigma, welche Sprache sollte ich nutzen?

-  Peter J. Landin. *The next 700 Programming Languages*. Communications of the ACM 9(3):157-166, 1966.
-  Jeffrey S. Foster. *Shedding New Light on an Old Language Debate*. Communications of the ACM 60(10):90, 2017.
-  Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, Vladimir Filkov. *A Large-Scale Study of Programming Languages and Code Quality in GitHub*. Communications of the ACM 60(10):91-100, 2017.
-  Rachel Harrison L. G. Smaraweera, Mark R. Dobie, Paul H. Lewis. *Comparing Programming Paradigms: An Evaluation of Functional and Object-Oriented Programs*. Software Engineering Journal 11(4):247-254, 1996.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-157/170

Quis leget haec?  
Wer soll das lesen?

Persius (34 - 62 n.Chr.)  
röm. Dichter

Non omnia possumus omnes!  
Wir können nicht alle alles!  
Wir können alle nicht alles!  
Wir alle können nicht alles!

Vergil (70 - 19 v.Chr.)  
röm. Dichter und Epiker

Nicht einmal alles lesen.  
Jeder aber kann etwas,  
auch einiges lesen.  
Und sollte das.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

158/170

Lege multum, non multa!  
Lies viel, nicht vieles!

Plinius der Jüngere (um 61 - um 113 n.Chr.)  
röm. Beamter und Schriftsteller  
beschrieb den Ausbruch des Vesuvs im Jahr 79 n.Chr.

Drei Vorschläge:

- ▶ Sie sind mit objektorientierter Programmierung groß geworden und fühlen sich dort heimisch?

Haskell: Eine Einführung für Objektorientierte von Ernst-Erich Doberkat könnte Ihre Wahl sein.

- ▶ Sie möchten die Welt funktionaler Programmierung zugleich mit Beispielen weiterer funktionaler Sprachen erkunden?

Funktionale Programmierung in OPAL, ML, Haskell und Gofer von Peter Pepper bietet sich als Ihre Wahl an.

- ▶ Sie suchen ein Buch, das möglichst weite Teile der Vorlesung überstreicht?

Haskell: The Craft of Functional Programming von Simon Thompson könnte sich für Sie lohnen.

Inhalt

Teil I

Kap. 1

1.1

1.2

1.3

1.4

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

159/170

# Teil II

## Grundlagen

Inhalt

Teil I

Kap. 1

**Teil II**

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Kapitel 2

## Elementare Typen, Tupel, Listen, Zeichenreihen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Kapitel 2.1

## Elementare Typen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

**2.1**

2.1.1

2.1.2

2.1.3

2.1.4

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

162/170

# Überblick

## Elementare (Daten-) Typen

- ▶ Wahrheitswerte: Bool
- ▶ Ganze Zahlen: Int, Integer
- ▶ Gleitkommazahlen: Float, Double
- ▶ Zeichen: Char

...in der Folge nach folgendem Schema angegeben:

- ▶ Typname
- ▶ Typtypische Konstanten
- ▶ Typtypische Operatoren und Relatoren

Für Details siehe [Haskell-Sprachbericht](#) und [Standard-Präliminam](#).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.1.1

2.1.2

2.1.3

2.1.4

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

163/170

# Kapitel 2.1.1

## Wahrheitswerte

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

**2.1.1**

2.1.2

2.1.3

2.1.4

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Wahrheitswerte: Bool

Typ	Bool	Wahrheitswerte
Konstanten	<code>True :: Bool</code> <code>False :: Bool</code>	Darstellung v. 'wahr' Darstellung v. 'falsch'
Vordef. Name	<code>otherwise :: Bool</code> <code>otherwise = True</code>	(Bedingte Ausdrücke: Stets erfüllter Wächter)
Operatoren	<code>(&amp;&amp;) :: Bool -&gt; Bool -&gt; Bool</code> <code>(  ) :: Bool -&gt; Bool -&gt; Bool</code> <code>not :: Bool -&gt; Bool</code>	Konjunktion Disjunktion Negation
Relatoren	<code>(==) :: Bool -&gt; Bool -&gt; Bool</code> <code>(/=) :: Bool -&gt; Bool -&gt; Bool</code> <code>(&gt;) :: Bool -&gt; Bool -&gt; Bool</code> ...	gleich ungleich echt größer

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.1.1

2.1.2

2.1.3

2.1.4

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

165/170

# Kapitel 2.1.2

## Ganze Zahlen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.1.1

**2.1.2**

2.1.3

2.1.4

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Ganze Zahlen: Int, Integer (1)

Typ

Int

Ganze Zahlen,  $-2^{63}$   
bis  $2^{63} - 1$  oder  $-2^{31}$   
bis  $2^{31} - 1$  (engl.  
fixed-precision integers)

Konstanten

0 :: Int

42 :: Int

-5 :: Int

...

Null

Zweiundvierzig

Minus fünf

Operatoren

(+) :: Int -> Int -> Int

(\*) :: Int -> Int -> Int

(^) :: Int -> Int -> Int

(-) :: Int -> Int -> Int

- :: Int -> Int

div :: Int -> Int -> Int

mod :: Int -> Int -> Int

abs :: Int -> Int

negate :: Int -> Int

...

fromInt :: Num a => Int -> a

Addition

Multiplikation

Exponentiation

Subtraktion

Vorzeichenwechsel

Division

Divisionsrest

Absolutbetrag

Vorzeichenwechsel

Typkonversion

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.1.1

2.1.2

2.1.3

2.1.4

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

167/170

# Ganze Zahlen: Int, Integer (2)

Relatoren	<code>(==)</code>	<code>:: Int -&gt; Int -&gt; Bool</code>	gleich
	<code>(/=)</code>	<code>:: Int -&gt; Int -&gt; Bool</code>	ungleich
	<code>(&gt;=)</code>	<code>:: Int -&gt; Int -&gt; Bool</code>	größer oder gleich
	<code>(&gt;)</code>	<code>:: Int -&gt; Int -&gt; Bool</code>	echt größer
	<code>(&lt;=)</code>	<code>:: Int -&gt; Int -&gt; Bool</code>	kleiner oder gleich
	<code>(&lt;)</code>	<code>:: Int -&gt; Int -&gt; Bool</code>	echt kleiner
	...		

**Vorgriff:** Numerische Typen in Haskell: Ganze Zahlen, Gleitkommazahlen, rationale und komplexe Zahlen, zusammengefasst in der Typklasse `Num` (vgl. `fromInt :: Num a => Int -> a`)



# Ganze Zahlen: Int, Integer (3)

Typ

Integer

Ganze Zahlen, keine Bereichsbeschränkung (engl. arbitrary-precision integers)

Konstanten

```
0 :: Integer
42 :: Integer
-5 :: Integer
93948307853803234 :: Integer
...
```

Null  
Zweiundvierzig  
Minus fünf  
'Große' Zahl

Operatoren

```
(+) ::
    Integer -> Integer -> Integer
...
fromInteger ::
    Num a => Integer -> a
```

Addition

Typkonversion

Relatoren

```
...
```

Konstanten, Operatoren und Relatoren für `Integer` wie für `Int`, jedoch keine *a-priori* Zahlbereichsbeschränkung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.1.1

2.1.2

2.1.3

2.1.4

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

169/170

# Kapitel 2.1.3

## Gleitkommazahlen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.1.1

2.1.2

**2.1.3**

2.1.4

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

170/170

# Gleitkommazahlen: Float, Double (1)

Typ	Float	Gleitkommazahlen (32 Bit-Darstellung)
Konstanten	0.125 :: Float -1.75 :: Float 8.5e-2 :: Float ...	Ein Achtel Minus eindreiviertel Achteinhalbhundertstel
Operatoren	(+) :: Float -> Float -> Float (*) :: Float -> Float -> Float ... sqrt :: Float -> Float  sin :: Float -> Float cos :: Float -> Float ... ceiling :: Float -> Int floor :: Float -> Int round :: Float -> Int	Addition Multiplikation  (positive) Quadrat- wurzel sinus cosinus  Typkonversion Typkonversion Typkonversion

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1.1

2.1.2

2.1.3

2.1.4

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Gleitkommazahlen: Float, Double (2)

Relatoren	(==) :: Float -> Float -> Bool	gleich
	(/=) :: Float -> Float -> Bool	ungleich
	(>=) :: Float -> Float -> Bool	größer oder gleich
	(>) :: Float -> Float -> Bool	echt größer
	(<=) :: Float -> Float -> Bool	kleiner oder gleich
	(<) :: Float -> Float -> Bool	echt kleiner
	...	

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.1.1

2.1.2

2.1.3

2.1.4

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

172/170

# Gleitkommazahlen: Float, Double (3)

Typ	Double	Gleitkommazahlen (64 Bit-Darstellung)
Konstanten	...	
Operatoren	...	
Relatoren	...	

Konstanten, Operatoren und Relatoren für Double wie für Float, jedoch mit doppelter Genauigkeit.

# Kapitel 2.1.4

## Zeichen, Ziffern, Sonderzeichen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.1.1

2.1.2

2.1.3

**2.1.4**

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

174/170

# Zeichen: Char

Typ	Char	Zeichen (Literal) (Unicode-Darst.)	Inhalt
Konstanten	'a' :: Char	Darst. von a	Teil I Kap. 1
	...		Teil II
	'Z' :: Char	Darst. von Z	Kap. 2
	'\t' :: Char	Tabulator	2.1
	'\n' :: Char	Neue Zeile	2.1.1
	'\\' :: Char	'backslash'	2.1.2
	'\"' :: Char	Hochkomma	2.1.3
Operatoren	'\"' :: Char	Anführungszeichen	2.1.4
	...		2.2
	ord :: Char -> Int	Konversionsfkt.	2.3
	chr :: Int -> Char	Konversionsfkt.	2.4
Relatoren	...		2.5
	(==) :: Char -> Char -> Bool	gleich	Kap. 3
	(>) :: Char -> Char -> Bool	echt größer	Kap. 4
	...		Kap. 5

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 2.2

## Tupel

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

**2.2**

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11



# Tupel

## Tupel

- ▶ fassen eine **vorbestimmte** Zahl von Werten möglicherweise **verschiedener** Typen zusammen.
- ▶ sind in diesem Sinn **heterogen**.

# Allgemeines Muster

## ► Allgemeines Muster für Tupelwerte

$(v1, v2, \dots, vk) :: (T1, T2, \dots, Tk)$

Dabei bezeichnen  $v1, \dots, vk$  Werte und  $T1, \dots, Tk$  Typen mit

$v1 :: T1, v2 :: T2, \dots, vk :: Tk$

## ► Standardkonstruktor (runde Klammern)

### ► Leerer Tupelkonstruktor:

$() :: ()$  (exotisch, aber sinnvoll, s. Kap. 15)

### ► Paarkonstruktor:

$(,) :: a \rightarrow b \rightarrow (a, b)$

### ► Tripelkonstruktor:

$(,,) :: a \rightarrow b \rightarrow c \rightarrow (a, b, c)$

### ► Quadrupelkonstruktor:

$(,,, ) :: a \rightarrow b \rightarrow c \rightarrow d \rightarrow (a, b, c, d)$

### ► ...

# Beispiele für Tupel

## Beispiele:

```
(,) 3.14 17.4 ->> (3.14,17.4) :: (Float,Float)
(,) 'a' True   ->> ('a',True)  :: (Char,Bool)
(,) "Fun" 3    ->> ("Fun",3)   :: (String,Int)
(,) ("Fun",3) True
    ->> (("Fun",3),True) :: ((String,Int),Bool)
(,,) 5 8 6.5 ->> (5,8,6.5) :: (Int,Int,Float)
(,,,) 'b' False "Fun" 3
    ->> ('b',False,"Fun",3) :: (Char,Bool,String,Int)
p = (,,,) 'b' False :: a -> b -> (Char,Bool,a,b)
p "Fun" 3
    ->> ('b',False,"Fun",3) :: (Char,Bool,String,Int)
p 2.1 True
    ->> ('b',False,2.1,True) :: (Char,Bool,Float,Bool)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Standardselektoren für Paare

**Standardselektoren** (vordefiniert ausschließlich für **Paare**):

```
fst :: (a,b) -> a
```

```
fst (x,_) = x
```

```
snd :: (a,b) -> b
```

```
snd (_,y) = y
```

**Aufrufbeispiele:**

```
fst (3.14,'a') ->> 3.14
```

```
snd (3.14,'a') ->> 'a'
```

**Bemerkung:** Für **drei- und mehrstellige Tupel** bietet Haskell keine vordefinierten (Selektor-) Funktionen für den Zugriff auf Tupelkomponenten an.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Selektoren für Tripel

## Selbstdefinierte Selektoren für Tripel:

$\text{fst}' :: (a,b,c) \rightarrow a$

$\text{fst}' (x,_,_) = x$

$\text{snd}' :: (a,b,c) \rightarrow b$

$\text{snd}' (_,y,_) = y$

$\text{thd}' :: (a,b,c) \rightarrow c$

$\text{thd}' (_,_,z) = z$

## Aufrufbeispiele:

$\text{fst}' (3.14, 'a', \text{True}) \rightarrow 3.14$

$\text{snd}' (3.14, 'a', \text{True}) \rightarrow 'a'$

$\text{thd}' (3.14, 'a', \text{True}) \rightarrow \text{True}$

# Kapitel 2.3

## Listen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

**2.3**

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Listen

## Listen

- ▶ fassen eine **nicht vorbestimmte** Zahl von Werten **gleichen** Typs zusammen.
- ▶ sind in diesem Sinn **homogen**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

**2.3**

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Allgemeines Muster

## ► Allgemeines Muster für Listenwerte

$v1 : (v2 : (\dots : (vk : [])) \dots) = [v1, v2, \dots, vk] :: [T]$

Dabei bezeichnen  $v1, \dots, vk$  Werte und  $T$  einen Typ mit  
 $v1, v2, \dots, vk :: T$

## ► Standardkonstruktor $(:)$ , zusätzlich eckige Klammern als Listenoperator für kompaktere Schreibweise

### ► Allgemein:

$v1 : (v2 : (\dots : (vk : [])) \dots) :: [a]$  (Standard)  
 $[v1, v2, \dots, vk] :: [a]$  (Abgekürzt)

### ► Konkret:

$1 : (2 : (3 : (4 : []))) :: [Int]$  (Standard)  
 $[1, 2, 3, 4] :: [Int]$  (Abgekürzt)  
 $[] :: [a]$  (Leere Liste)

Erinnerung: `quickSort [] = []`  
`quickSort (n:ns) = ...`



# Beispiele für Listen (1)

...von

- ▶ Zeichen

`['a','b','c','d','e'] :: [Char]`

- ▶ Wahrheitswerten

`[True,False,True] :: [Bool]`

- ▶ ganzen Zahlen

`[2,5,17,4,42,4711] :: [Int]`

- ▶ Gleitkommazahlen

`[3.14,5.0,12.21] :: [Float]`

- ▶ keinen Elementen (bel. Typs), leere Liste (bel. Typs)

`[] :: [a]`

# Beispiele für Listen (2)

...von

## ► Tupeln

```
[('a',True),('b',False),('c',False),  
 ('d',False),('e',True)] :: [(Char,Bool)]  
  
[(3,5,4.0),(4,7,5.5),(2,8,5.0),(2,11,6.5)]  
                                :: [(Int,Int,Float)]
```

## ► Listen

```
[[1,2,3],[9],[],[17,4,21],[],[3,2]] :: [[Int]]  
  
[(['f','p'],2),(['h'],1),([],[0])] :: [([Char],Int)]  
  
[("fun",3),("h",1),("",0)] :: [([Char],Int)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Beispiele für Listen (3)

...von

## ► Zeichenreihen

```
["sin","cos","tan","sin","cos","tan"] :: [[Char]]
```

## ► Funktionen

```
[sin,cos,tan,sin,cos,tan] :: [Float -> Float]
```

```
[(+),(*),ggt,mod] :: [Int -> Int -> Int]
```

```
[binom',binom''] :: [(Integer,Integer) -> Integer]
```

```
[binom,binom] :: [Integer -> Integer -> Integer]
```

## ► ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Vergleichbarkeit und Gleichheit von Listen (1)

Nur **typgleiche** Listenwerte sind **vergleichbar**:

```
cs = ['a','b','c'] :: [Char]
ss = ["a","b","c"] :: [String]
xs = "abc" :: String

ns = [1,2,3] :: [Int]
ms = [1,2,3] :: [Integer]

fs = [1.0,2.0,3.0] :: [Float]
ds = [1.0,2.0,3.0] :: [Double]

cs == ns ->> "Fehler: Typfehler in Anwendung"
ns == fs ->> "Fehler: Typfehler in Anwendung"
ns == ms ->> "Fehler: Typfehler in Anwendung"
fs == ds ->> "Fehler: Typfehler in Anwendung"

ns == ns ->> True
fs == fs ->> True

cs == ss ->> "Fehler: Typfehler in Anwendung"
ss == xs ->> "Fehler: Typfehler in Anwendung"

cs == xs ->> True
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
188/170

# Vergleichbarkeit und Gleichheit von Listen (2)

Nur **typgleiche** Listen gleicher Länge u. Anordnung sind gleich:

```
ns = [1,2,3,4]
```

```
ms = [1,2,3]
```

```
ks = [1,2,3,4,5]
```

```
ls = [2,1,4,3]
```

```
ns == ns ->> True
```

```
ns == ms ->> False
```

```
ns == ks ->> False
```

```
ns == ls ->> False
```

```
[17,4,21] :: [Int] == [17,4,21] :: [Int] ->> True
```

```
[17,4,21] :: [Int] == [17,4,21] :: [Integer] ->>  
"Fehler: Typfehler in Anwendung"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
189/170

# Vergleichbarkeit und Gleichheit von Listen (3)

Auch 'leere' Listen sind nur **typabhängig** vergleichbar:

```
[] == [] ->> True  
[] :: [Int] == [] :: [Int] ->> True  
[] :: [Int] == [] :: [Integer]  
    ->> "Fehler: Typfehler in Anwendung"  
[] :: [Float] == [] :: [Double]  
    ->> "Fehler: Typfehler in Anwendung"  
bs = [] :: [Bool]  
cs = [] :: [Char]  
bs == cs ->> "Fehler: Typfehler in Anwendung"  
bs /= cs ->> "Fehler: Typfehler in Anwendung"  
xs = []  
ys = []  
xs == ys ->> True  
xs /= ys ->> False
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Vordefinierte Funktionen auf Listen (1)

## Name und Typ

## Bedeutung und Beispiel

`(:)` `:: a -> [a] -> [a]`

Anfügen eines Elements am Anfang einer Liste:

`5 : [3,2] ->> [5,3,2]`

`(++)` `:: [a] -> [a] -> [a]`

Aneinanderhängen zweier Listen:

`[11,7] ++ [5,3,2] ->> [11,7,5,3,2]`

`(!!)` `:: [a] -> Int -> a`

Zugreifen auf ein Listenelement:

`[5,3,2] !! 0 ->> 5`

`[5,3,2] !! 1 ->> 3`

`concat` `:: [[a]] -> [a]`

Verschmelzen einer Liste von Listen zu einer Liste:

`concat [[11,7], [5,3,2]]`  
`->> [11,7,5,3,2]`

`reverse` `:: [a] -> [a]`

Umkehren einer Liste:

`reverse [5,3,2] ->> [2,3,5]`

...und viele mehr (siehe [Standard-Präludium](#)).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
191/170

# Vordefinierte Funktionen auf Listen (2)

...drei Beispiele **vordefinierter Funktionen auf Listen** mit ihrer **Implementierung**.

Die Funktion **length**:

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + length xs
```

**Aufrufbeispiele:**

```
length [1,2,3]           ->> 3
length [[1],[2,3],[4,5,6]] ->> 3
length [sin,cos,tan]     ->> 3
length []                 ->> 0
```



# Vordefinierte Funktionen auf Listen (3)

Die Funktionen `head` und `tail`:

```
head :: [a] -> a
```

```
head (x:_) = x
```

```
tail :: [a] -> [a]
```

```
tail (_:xs) = xs
```

Aufrufbeispiele:

```
head [1,2,3] ->> 1
```

```
head [[1],[2,3],[4,5,6]] ->> [1]
```

```
head [sin,cos,tan] ->> sin
```

```
head [sin,cos,tan] (pi/2) ->> 1.0
```

```
tail [1,2,3] ->> [2,3]
```

```
tail [[1],[2,3],[4,5,6]] ->> [[2,3],[4,5,6]]
```

```
tail [sin,cos,tan] ->> [cos,tan]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
193/170

# Listenkomprehension

...Zusammenfassung, Vereinigung von Mannigfaltigkeiten zu einer Einheit (Philos.):

- In funktionalen Sprachen ein wichtiges, ausdruckskräftiges Sprachkonstrukt, das eine automatische Listengenerierung ermöglicht!

Beispiele:

```
ns = [1..10]  -- kurz für [1,2,3,4,5,6,7,8,9,10]
```

```
[3*n | n <- ns] ->> [3,6,9,12,15,18,21,24,27,30]
```

```
[n | n <- ns, odd(n)] ->> [1,3,5,7,9]
```

```
[n | n <- ns, even(n), n > 5] ->> [6,8,10]
```

```
[n*(n+1) | n <- ns, (even(n) || n > 5)]  
->> [6,20,42,56,72,90,110]
```

```
[p | n <- ns, m <- ns, n <= 3, m >= 9, let p=m*n]  
->> [9,10,18,20,27,30]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
194/170

# Aufzählungsausdrücke

...stellen einen weiteren **Erzeugungsautomatismus** für **Listen** über Typen geordneter **aufzählbarer Werte** (Typen der Typklasse **Enum**) dar:

[2..13] ->> [2,3,4,5,6,7,8,9,10,11,12,13]

[2,5..20] ->> [2,5,8,11,14,17,20]

[2,5..21] ->> [2,5,8,11,14,17,20]

[2,5..22] ->> [2,5,8,11,14,17,20]

[2,5..23] ->> [2,5,8,11,14,17,20,23]

[11,9..3] ->> [11,9,7,5,3]

[11,9..2] ->> [11,9,7,5,3]

[11,10..2] ->> [11,10,9,8,7,6,5,4,3,2]

[11..2] ->> []

['a','c'..'g'] ->> ['a','c','e','g'] ->> "aceg"

['a','c'..'h'] ->> ['a','c','e','g'] ->> "aceg"

[0.0,0.3..1.2] ->> [0.0,0.3,0.6,0.9,1.2]

# Beachte

...folgende Gleichheiten und abkürzende Schreibweisen:

<code>(1:(2:(3:[])))</code>	(Standarddarstellung)
<code>== 1:2:3:[]</code>	(Rechtsassoziativität)
<code>== [1,2,3]</code>	(Syntaktischer Zucker)

...Typisierungen und überladene Schreibweisen:

<code>[] :: [a]</code>	
<code>[1,2,3]</code>	
<code>ns = [1,2,3] :: [Int]</code>	
<code>ms = [1,2,3] :: [Integer]</code>	
<code>[1,2,3] :: Num a =&gt; [a]</code>	(Gleiche Schreibweise,
<code>ns :: [Int]</code>	verschiedene Typen,
<code>ms :: [Integer]</code>	Überladung von [1,2,3])

...überprüfbar in Hugs mittels des Kommandos `:t`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
196/170

# Kapitel 2.4

## Zeichenreihen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

**2.4**

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Zeichenreihen

## Zeichenreihen (engl. **Strings**)

- ▶ fassen eine **nicht vorbestimmte** Zahl von Werten des Typs Zeichen **Char** zusammen.
- ▶ sind spezielle Listen und deshalb ebenfalls **homogen**.

# Zeichenreihen: String

Zeichenreihen sind in Haskell über dem Datentyp Liste realisiert, als Listen von Zeichen, kurz Zeichenlisten:

Typ	[Char]	Zeichenlisten
Bezeichner	String	Typsynonym
Vereinbarung	<pre>'data [a] = []   a:[a]       deriving (Eq,Ord)' type String = [Char]</pre>	Kein zulässiges Haskell; nur zur Illustration
Konstanten	<pre>['F','u','n'] :: String "Fun" :: String [] :: String "" :: String</pre>	Zwei Darst. der Z-Reihe 'Fun' und der leeren Z-Reihe
Operatoren	<pre>(++) :: String -&gt; String -&gt; String ...</pre>	Konkatenation
Relatoren	<pre>(==) :: String -&gt; String -&gt; Bool (/=) :: String -&gt; String -&gt; Bool</pre>	gleich ungleich

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Beispiele mit Zeichenreihen

...zu Konstruktion und Gleichheitstests:

```
"Fun" == ['F','u','n'] ->> True
```

```
['F','u','n'] == ('F':('u':('n':[]))) ->> True
```

```
"Fun" == ('F':('u':('n':[]))) ->> True
```

```
['H','e','l','l','o'] ++ ", " ++ " " ++ "world!"  
== "Hello, world!" ->> True
```

```
"a" == 'a' ->> "Fehler: Typfehler in Anwendung"
```

```
"a" == ['a'] ->> True
```

```
"a" == [] ->> False
```

```
'a' == [] ->> "Fehler: Typfehler in Anwendung"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
200/170



# Vordefinierte Funktionen auf Zeichenreihen

...Zeichenreihen sind Listen über dem Zeichentyp Char, mithin Listen:

```
type String = [Char]
```

Deshalb stehen alle auf Listen vordefinierte Operatoren und Relatoren unmittelbar auch auf Zeichenreihen zur Verfügung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Kapitel 2.5

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

**2.5**

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8





Kap. 9

Teil IV




Kap. 10

Kap. 11

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 2 (1)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 2, Einfache Datentypen; Kapitel 5.1, Listen; Kapitel 5.2, Tupel; Kapitel 5.3, Zeichenreihen)
-  Richard Bird. *Introduction to Functional Programming using Haskell*. Cambridge University Press, 2. Auflage, 1998. (Kapitel 2, Simple datatypes; Kapitel 4, Lists)
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 2, Expressions, types, and values; Kapitel 4, Lists)
-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 1, Elemente funktionaler Programmierung)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 2 (2)

-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3.1, Basic concepts; Kapitel 3.2, Basic types; Kapitel 3.3, List types; Kapitel 3.4, Tuple types; Kapitel 5, List comprehensions)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 1, An Intro to Lists, Tuples; Kapitel 2, Common Haskell Types)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 2, Types and Functions – Useful Composite Data Types: Lists and Tuples, Functions over Lists and Tuples)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8



Kap. 9

Teil IV

Kap. 10

Kap. 11

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 2 (3)

-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 3.2, Elementare Strukturen; Kapitel 15, Listen (Sequenzen))
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 3, Basic types and definitions; Kapitel 5, Data types, tuples and lists)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

2.1

2.2

2.3

2.4

2.5

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Kapitel 3

## Funktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

**Kap. 3**

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Kapitel 3.1

## Definition, Schreibweisen, Sprachkonstrukte

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Funktionen

...sind wichtigstes **Abstraktions-** und **Ausdrucks**mittel in funktionaler Programmierung.

Funktionale Programmiersprachen bieten deshalb oft mehrere **Schreibweisen** an, um Funktionen zu definieren. So ist

```
fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n-1)
```

nur eine Möglichkeit, die **Fakultätsfunktion in Haskell** zu definieren:

$$\begin{aligned} & ! : \mathbb{IN} \rightarrow \mathbb{IN} \\ n! &= \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)! & \text{sonst} \end{cases} \end{aligned}$$



# Prägnanz d. Vermeiden bedingter Ausdrücke

Haskell bietet weitere Schreibweisen an, die meist **knapper**, **konziser** und deshalb **übersichtlicher** und **verständlicher** sind, insbesondere durch weitere Möglichkeiten

## ► Fallunterscheidungen

anders als durch **bedingte Ausdrücke** wie in

```
fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n-1)
           Bedingter Ausdruck
```

auszudrücken, insbesondere

## ► bewachte Ausdrücke

## ► Muster

# Schreibweisen für Funktionsdefinitionen (1)

## (1) Mittels bedingter Ausdrücke:

```
fac :: Int -> Int
```

```
fac n = if n == 0 then 1 else n * fac (n-1)  
      Bedingter Ausdruck
```

## (2) Mittels bewachter Ausdrücke:

```
fac :: Int -> Int
```

```
fac n
```

```
| n == 0           = 1  
  Wächter         Bewachter Ausdruck
```

```
| otherwise = n * fac (n-1)  
  Wächter   Bewachter Ausdruck
```

(otherwise, der  
stets erfüllte  
Wächter)

```
otherwise :: Bool
```

```
otherwise = True
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

210/170

# Schreibweisen für Funktionsdefinitionen (2)

(3a) Mittels Muster (hier für ganze Zahlen):

```
fac :: Int -> Int           -- Fakultätsfunktion
fac 0 = 1
fac n = n * fac (n - 1)
```

```
fib :: Int -> Int           -- Fibonacci-Funktion
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

211/170

# Schreibweisen für Funktionsdefinitionen (3)

## (3b) Mittels Muster (hier für Zeichen):

```
capitalizeVowels :: Char -> Char  
capitalizeVowels = capVow
```

```
capVow :: Char -> Char  
capVow 'a' = 'A'  
capVow 'e' = 'E'  
capVow 'i' = 'I'  
capVow 'o' = 'O'  
capVow 'u' = 'U'  
capVow c  = c
```

## (3c) Mittels Muster (hier für Wahrheitswerte):

```
xor :: Bool -> Bool -> Bool  
xor True  False = True  
xor False True  = True  
xor b1    b2    = False
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

212/1700

# Schreibweisen für Funktionsdefinitionen (4)

(3d) Mittels Muster (hier für Listen):

```
quickSort :: [Integer] -> [Integer]
```

```
quickSort [] = []
```

*Muster leere Liste*

```
quickSort (n : ns)
```

*Muster Listenkopf* *Muster Listenrest*

*Muster nichtleere Liste*

```
= quickSort [m | m <- ns, m <= n]
```

```
++ [n]
```

```
++ quickSort [m | m <- ns, m > n]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

213/170

# Schreibweisen für Funktionsdefinitionen (5)

(3e) Mittels **Muster**, hier zusätzlich mit 'wild card'-Muster:

```
mult :: Int -> Int -> Int
```

```
mult 0 _ = 0
```

```
mult _ 0 = 0
```

```
mult 1 y = y
```

```
mult x 1 = x
```

```
mult x y = x*y
```

```
tail :: [a] -> [a]
```

```
tail (_:xs) = xs
```

```
tail _      = error "Liste darf nicht leer sein."
```

```
nand :: Bool -> Bool -> Bool
```

```
nand False _ = True
```

```
nand _ False = True
```

```
nand _ _      = False      (auch in xor ist _ mögl.)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

214/170

# Schreibweisen für Funktionsdefinitionen (6)

Mittels Muster und...

(4) **case**-Ausdrucks:

```
capVow :: Char -> Char
```

```
capVow c = case c of 'a' -> 'A'  
                    'e' -> 'E'  
                    'i' -> 'I'  
                    'o' -> 'O'  
                    'u' -> 'U'  
                    otherwise -> c
```

```
describeList :: [a] -> String
```

```
describeList ls  
  = "The list ls "  
    ++ case ls of []      -> "is empty."  
           (x:[])      -> "is a singleton list."  
           (x:y:[])    -> "has two elements."  
           _          -> "has three or more elements."
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

215/1700

# Schreibweisen für Funktionsdefinitionen (7)

Mittels Muster und...

(5a) lokaler Deklarationen (**where**-Konstrukt, nachgestellt):

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
quickSort (n:ns) = quickSort smaller
                  ++ [n]
                  ++ quickSort larger
  where
    smaller = [m | m <- ns, m <= n]
    larger  = [m | m <- ns, m > n]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

216/170



# Schreibweisen für Funktionsdefinitionen (8)

Mittels Muster und...

(5b) lokaler Deklarationen (**let**-Konstrukt, vorgestellt):

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
quickSort (n:ns) = let
                    smaller = [m | m<-ns, m<=n]
                    larger  = [m | m<-ns, m>n]
in (quickSort smaller
    ++ [n]
    ++ quickSort larger)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

217/1700

# Schreibweisen für Funktionsdefinitionen (9)

In einer Zeile mittels **where**-Konstrukts und...

## (6a) Semikolons ';':

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
quickSort (n:ns) =
    quickSort smaller ++ [n] ++ quickSort larger
    where smaller = [m | m <- ns, m <= n]; larger = [m | m <- ns, m > n]
```

In einer Zeile mittels **let**-Konstrukts und...

## (6b) Semikolons ';':

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
quickSort (n:ns) =
    let smaller = [m | m <- ns, m <= n]; larger = [m | m <- ns, m > n]
    in (quickSort smaller ++ [n] ++ quickSort larger)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

218/1700

# Schreibweisen für Funktionsdefinitionen (10)

## (7a) Mittels anderer Funktionen (argumentbehaftet):

```
fac :: Int -> Int
```

```
fac n = foldl (*) 1 [1..n]
```

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

```
foldl f z [] = z
```

```
foldl f z (x:xs) = foldl f (f z x) xs
```

```
fac :: Int -> Int
```

```
fac n = product [1..n]
```

```
product :: (Num a) => [a] -> a
```

```
product ns = foldl (*) 1 ns -- argumentbehaftet
```

```
-- Gleichbedeutend:
```

```
product = foldl (*) 1 -- argumentfrei
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

219/1700

# Schreibweisen für Funktionsdefinitionen (11)

(7b) Mittels anderer Funktionen (argumentfrei):

```
factorial :: Int -> Int
```

```
factorial = fac
```

```
qs :: [Integer] -> [Integer]
```

```
qs = quickSort
```

...wenn z.B. der Name `fac` zu wenig sprechend, der Name `quickSort` zu lang erscheint (vgl. auch das Funktionenpaar `capitalizeVowels` und `capVow`).

# Schreibweisen für Funktionsdefinitionen (12)

(7b) Mittels anderer Funktionen (argumentfrei), fgs.:

```
and :: Bool -> Bool -> Bool  
and = (&&)
```

Zusätzlich zu Ausdrücken der Form:

$(x > 0) \ \&\& \ (y < 0)$        $(\&\&) \ (x > 0) \ (y < 0)$

sind nun auch Ausdrücke der Form:

$(x > 0) \ 'and' \ (y < 0)$        $and \ (x > 0) \ (y < 0)$

möglich!

```
(./.) :: Integer -> Integer -> Integer
```

```
(./.) = div
```

Zusätzlich zu Ausdrücken der Form:

$div \ 5 \ 2$        $5 \ 'div' \ 2$

sind nun auch Ausdrücke der Form:

$5 \ ./.\ 2$        $(./.) \ 5 \ 2$

möglich!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

221/170

# Schreibweisen für Funktionsdefinitionen (13)

## (8) Mittels anonymer Funktionen (argumentfrei):

```
fac = \n -> (if n == 0 then 1 else n * fac (n-1))
```

*Anonyme  $\lambda$ -Abstraktion*

## Die Schreibweise ist

- ▶ Reminiszenz an den funktionaler Programmierung zugrundeliegenden  $\lambda$ -Kalkül:
  - ▶ Im  $\lambda$ -Kalkül:  $\lambda x y. x + y$
  - ▶ In Haskell: `\x y -> x+y`

## Anwendung in Haskell:

- ▶ Immer dann, wenn der Funktionsname keine Rolle spielt:

```
map (\n -> 2*n+1) [1,2,3] ->> [3,5,7]
map (\n -> n*n-1) [1,2,3] ->> [0,3,8]
```

# Verwendungshinweise: Bewachte Ausdrücke (1)

Funktionen sind außer in den einfachsten Fällen fast immer über Fallunterscheidungen definiert.

- Bewachte Ausdrücke führen meist zu besserer Lesbarkeit als (geschachtelte) bedingte Ausdrücke.

Vergleiche:

<code>signum :: Int -&gt; Int</code>	<code>signum' :: Int -&gt; Int</code>
<code>signum n</code>	<code>signum' n</code>
<code>  n &lt; 0 = -1</code>	<code>  n &lt; 0 = -1</code>
<code>  n == 0 = 0</code>	<code>  n == 0 = 0</code>
<code>  n &gt; 0 = 1</code>	<code>  otherwise = 1</code>

(Ein Tick effizienter!)

mit:

```
signum'' :: Int -> Int
signum'' n = if n < 0 then -1 else
              if n == 0 then 0 else 1
```

# Verwendungshinweise: Bewachte Ausdrücke (2)

Mischformen sind möglich, aber mindestens auf die Weise wie hier gar nicht schön, sinnvoll oder empfehlenswert:

```
signum''' :: Int -> Int
signum''' n
  | n < 0      = -1
  | otherwise = if n == 0 then 0 else 1
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

224/170



# Verwendungshinweise: Muster (1)

Funktionen arbeiten häufig auf strukturierten Werten.

- **Musterbasierte** Definitionen sind meist am zweckmäßigsten und übersichtlichsten (weil sie die Struktur offenlegen und **Selektoren** dadurch unnötig machen).

Vergleiche:

```
binom' :: (Int,Int) -> Int
binom' (n,k)
  | k==0 || n==k = 1
  | otherwise    = binom' (n-1,k-1) + binom' (n-1,k)
```

mit:

```
binom'' :: (Int,Int) -> Int
binom'' p
  | snd(p) == 0 || fst(p) == snd(p) = 1
  | otherwise = binom'' (fst(p)-1,snd(p)-1)
                + binom'' (fst(p)-1,snd(p))
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

225/170

# Verwendungshinweise: Muster (2)

Vorteile musterbasierter Funktionsdefinitionen:

## Muster

- ▶ legen die **Struktur des (Argument-) Werts** offen
- ▶ legen **Namen** für die verschiedenen Strukturteile des Werts fest und erlauben über diese Namen **unmittelbaren Zugriff** auf diese Teile
- ▶ vermeiden dadurch sonst nötige **Selektorfunktionen**

und führen dadurch zu einem

- ▶ **Gewinn an Lesbarkeit und Transparenz.**

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

226/170

# Bezeichnungskonventionen für Muster (1)

Für `Int(eger)`-Werte: `n, m, ...`

für `Listen` von `Int(eger)`-Werten: `ns, ms, ...`

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
quickSort (n:ns) = quickSort [m | m <- ns, m <= n]
                  ++ [n]
                  ++ quickSort [m | m <- ns, m > n]
```

Für `Werte` beliebigen `Typs`: `x, y, ...`

für `Listenwerte` beliebigen `Typs`: `xs, ys, ...`

```
quickSort :: Ord a => [a] -> [a]
quickSort []      = []
quickSort (x:xs) = quickSort [y | y <- xs, y <= x]
                  ++ [x]
                  ++ quickSort [y | y <- xs, y > x]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

227/170

# Bezeichnungskonventionen für Muster (2)

- ▶ Für Zeichen-Werte:  $c, c', d, \dots$   
für Listen von Zeichen-Werten:  $cs, cs', ds, \dots$
- ▶ Für Ziffern-Werte:  $d, d', e, \dots$   
für Listen von Ziffern-Werten:  $ds, ds', es, \dots$
- ▶ Für Wahrheitswerte:  $b, b', \dots$   
für Listen von Wahrheitswerten:  $bs, bs', \dots$
- ▶ Für ganze Zahlen:  $n, n', m, \dots$   
für Listen ganzer Zahlen:  $ns, ns', ms, \dots$
- ▶ Für Werte beliebigen Typs:  $x, x', y, y', \dots$   
für Listen von Werten beliebigen Typs:  $xs, xs', ys, ys', \dots$
- ▶ Für Listen von Listen-Werten:  $nss, mss, xss, yss, \dots$
- ▶ ...

# Muster

...sind (soweit wie bislang eingeführt):

- ▶ **Konstanten** eines Typs (z.B. `0`, `42`, `3.14`, `False`, `True`, `'c'`, `"c"`, `"fun"`, `" "`, `[]`,...) ...ein Argumentwert passt mit dem Muster zusammen, wenn es wertgleich mit der Konstanten ist.
- ▶ **Variablen** (z.B. `n`, `x`, `c`,...) ...jeder Argumentwert passt.
- ▶ **Wild card** `'_'` ...jeder Argumentwert passt (Verwendung von `"_"` für alle Argumentwerte, die nicht zum Ergebnis beitragen; siehe `mult`, `tail`, `nand`,...).
- ▶ **Zusammengesetzte Muster**, bis jetzt für Tupel und Listen (z.B. `[]`, `[x]`, `(x:[])`, `(x:y:[])`, `(x:y:z:[])`, `(x:xs)`, `(x:y:xs)`, `(m,n)`, `(m,_)`, `(_,_)`, `(x,y,z)`, `(x,_,z)`,...)
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

229/1700

# Kapitel 3.2

## Funktionssignaturen, Funktionsterme, Funktionsstelligkeiten

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

**3.2**

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Überblick

## Funktions-

- ▶ Signaturen
- ▶ Terme
- ▶ Stelligkeiten

und damit verbundene

- ▶ Klammereinsparungsregeln in Haskell.

## Das Wichtigste auf einen Blick:

- ▶ (Funktions-) Signaturen sind rechtsassoziativ geklammert
- ▶ (Funktions-) Terme sind linksassoziativ geklammert
- ▶ (Funktions-) Stelligkeit ist 1

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

231/170

# Beispiel: Die Editorfunktion 'ersetze'

...eine Funktion, die in einem Text das  $n$ -te Vorkommen einer Zeichenreihe  $s$  durch eine Zeichenreihe  $s'$  ersetzt.

## Implementierung in Haskell

```
type Txt    = String
type Vork   = Int
type Alt    = Txt
type Neu    = Txt

ersetze :: (Txt -> (Vork -> (Alt -> (Neu -> Txt))))
```

...angewendet auf einen Text  $t$ , eine Vorkommensnummer  $n$  und zwei Zeichenreihen  $s$  und  $s'$  ist das Resultat der Anwendung von `ersetze` ein Text  $t'$ , in dem das  $n$ -te Vorkommen von  $s$  in  $t$  durch  $s'$  ersetzt ist.



# Eine Anwendung von 'ersetze'

## Funktion

```
type Txt  = String
type Vork = Int
type Alt  = Txt
type Neu  = Txt

ersetze :: (Txt -> (Vork -> (Alt -> (Neu -> Txt))))
```

## Argumente

```
"Ein alter Text" :: Txt
1 :: Vork
"alter" :: Alt
"neuer" :: Neu
```

## Auswertung

```
((((ersetze "Ein alter Text") 1) "alter") "neuer")
->> "Ein neuer Text" :: Txt
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

233/170

# Schrittweise Auswertung (1)

...**ersetze** und die nach fortgesetzter Argumentkonsumation entstehenden **Funktionsterme** sind mit Ausnahme des letzten von **funktionalem Typ**.

**Schritt 1:** Der Funktionsterm **ersetze** konsumiert **ein** Argument, den Wert **"Ein alter Text"** vom Typ **Txt**. Der dadurch entstehende **Funktionsterm** ist von funktionalem Typ:

```
(ersetze "Ein alter Text") ::  
      (Vork -> (Alt -> (Neu -> Txt)))
```

**Schritt 2:** Der Funktionsterm **(ersetze "Ein alter Text")** konsumiert **ein** Argument, den Wert **1** vom Typ **Vork**. Der dadurch entstehende **Funktionsterm** ist von funktionalem Typ:

```
((ersetze "Ein alter Text") 1) :: (Alt -> (Neu -> Txt))
```

# Schrittweise Auswertung (2)

**Schritt 3:** Der Funktionsterm `((ersetze "Ein alter Text") 1)` konsumiert **ein** Argument, den Wert `"alter"` vom Typ **Alt**. Der dadurch entstehende Funktionsterm ist von funktionalem Typ:

```
((ersetze "Ein alter Text") 1) "alter" :: (Neu -> Txt)
```

**Schritt 4:** Der Funktionsterm `((ersetze "Ein alter Text") 1) "alter"` konsumiert **ein** Argument, den Wert `"neuer"` vom Typ **Neu**. Der dadurch entstehende Term ist von **nichtfunktionalem** Typ:

```
((ersetze "Ein alter Text") 1) "alter" "neuer" :: Txt
```

Insgesamt erhalten wir:

```
((ersetze "Ein alter Text") 1) "alter" "neuer"  
->> "Ein neuer Text" :: Txt
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

235/170

# Funktionssignaturen, Funktionsterme

**Funktionssignaturen** (oder syntaktische Funktionssignaturen oder Signaturen)

- ▶ geben den **Typ einer Funktion** an.

**Funktionsterme**

- ▶ sind aus Funktionsaufrufen aufgebaute **Ausdrücke**.

Beispiele:

- ▶ **Funktionssignatur**

```
ersetze :: Txt -> Vork -> Alt -> Neu -> Txt
```

- ▶ **Funktionsterme**

```
ersetze "Ein alter Text"
```

```
ersetze "Ein alter Text" 1
```

```
ersetze "Ein alter Text" 1 "alter"
```

```
ersetze "Ein alter Text" 1 "alter" "neuer"
```

# Klammereinsparungsregeln

...für Funktionssignaturen und Funktionsterme.

**Rechtsassoziativität** für **Funktionssignaturen**:

`ersetze :: Txt -> Vork -> Alt -> Neu -> Txt`

...steht abkürzend für die vollständig, aber nicht überflüssig  
**rechtsassoziativ** geklammerte **Funktionssignatur**:

`ersetze :: (Txt -> (Vork -> (Alt -> (Neu -> Txt))))`

**Linksassoziativität** für **Funktionsterme**:

`ersetze "Ein alter Text" 1 "alter" "neuer"`

...steht abkürzend für den vollständig, aber nicht überflüssig  
**linksassoziativ** geklammerten **Funktionsterm**:

`((((ersetze "Ein alter Text") 1) "alter") "neuer")`

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

237/170

# Hintergrund

Die Festlegung von

- ▶ Rechtsassoziativität für Funktionssignaturen
- ▶ Linksassoziativität für Funktionsterme

dient der Einsparung von Klammern (vgl. Punkt- vor Strichrechnung in der Mathematik).

Die Festlegung erfolgt auf diese Weise, da so in

- ▶ Signaturen und Funktionstermen

meist möglichst wenige, oft gar keine Klammern nötig sind.

# 1-Stelligkeit von Funktionen in Haskell

Das **Beispiel** illustriert, dass Haskell-Funktionen **einstellig** sind:

- Es wird stets **ein** Argument zur Zeit konsumiert!

```
ersetze :: Txt -> (Vork -> Alt -> Neu -> Txt)
```

```
ersetze "Ein alter Text" :: Vork -> (Alt -> Neu -> Txt)
      :: Txt
```

```
:: Txt -> (Vork -> Alt -> Neu -> Txt)
```

(ersetze "Ein alter Text") 1 :: Alt -> (Neu -> Txt)

$:: \text{Vork} \rightarrow (\text{Alt} \rightarrow \text{Neu} \rightarrow \text{Txt}) \quad :: \text{Vork}$

$((\text{ersetze "Ein alter Text"})\ 1)\ \underbrace{\text{"alter"}}_{:: \text{Alt}} :: \text{Neu} \rightarrow \text{Txt}$

$((\text{ersetze "Ein alter Text"} \ 1) \ \text{"alter"}) \ \underbrace{\text{"neuer"}}_{:: \text{Neu}} :: \text{Txt}$

```
->> "Ein neuer Text" :: Txt
```

# Zur 1-Stelligkeit von Funktionen

Es gilt: Konsumierte Argumente müssen nicht elementar sein; ausgedrückt durch **Klammerung** können sie

- **zusammengesetzt** und **komplex** sein.

Beispiel:

$$\begin{array}{l} \text{add}' :: \underbrace{(\text{Int} \rightarrow \text{Int})}_{\text{'Arg. 1'}} \rightarrow \underbrace{(\text{Int} \rightarrow \text{Int})}_{\text{'Arg. 2'}} \rightarrow \underbrace{(\text{Int}, \text{Int})}_{\text{'Arg. 3'}} \rightarrow \underbrace{\text{Int}}_{\text{'Resultat'}} \\ \text{add}' \text{ f g (m,n) = (+) (f m) (g n)} \end{array}$$

Vollständig, aber nicht überflüssig geklammert:

$$\begin{array}{l} \text{add}' :: ((\text{Int} \rightarrow \text{Int}) \rightarrow ((\text{Int} \rightarrow \text{Int}) \rightarrow ((\text{Int}, \text{Int}) \rightarrow \text{Int}))) \\ \text{add}' \text{ f g (m,n) = (((+) (f m)) (g n))} \end{array}$$



# Beispiel 1: Konsumation komplexer Arg. (1)

$\text{add}' :: ((\text{Int} \rightarrow \text{Int}) \rightarrow ((\text{Int} \rightarrow \text{Int}) \rightarrow ((\text{Int}, \text{Int}) \rightarrow \text{Int})))$   
 $\text{add}' f g (m, n) = (((+) (f m)) (g n))$

$$\underbrace{(\text{add}' \text{ fac } \text{fib } (5, 7))}_{:: \text{Int}} \rightarrow$$
  
$$\underbrace{((\text{add}' \text{ fac}) \text{fib}) (5, 7)}_{:: ((\text{Int} \rightarrow \text{Int}) \rightarrow ((\text{Int}, \text{Int}) \rightarrow \text{Int}))} \rightarrow$$
  
$$\underbrace{((\text{Int} \rightarrow \text{Int}) \rightarrow ((\text{Int}, \text{Int}) \rightarrow \text{Int}))}_{:: ((\text{Int}, \text{Int}) \rightarrow \text{Int})} \rightarrow$$
  
$$\underbrace{((+) (f m))}_{:: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}} \quad \underbrace{(\text{fac } 5)}_{:: \text{Int}} \quad \underbrace{(\text{fib } 7)}_{:: \text{Int}}$$
  
$$\underbrace{((\text{Int} \rightarrow \text{Int}) \rightarrow ((\text{Int}, \text{Int}) \rightarrow \text{Int}))}_{:: \text{Int} \rightarrow \text{Int}} \quad \underbrace{((\text{Int}, \text{Int}) \rightarrow \text{Int})}_{:: \text{Int}}$$
  
$$\underbrace{((\text{Int} \rightarrow \text{Int}) \rightarrow ((\text{Int}, \text{Int}) \rightarrow \text{Int}))}_{:: \text{Int} \rightarrow \text{Int}}$$
  
$$\underbrace{((\text{Int} \rightarrow \text{Int}) \rightarrow ((\text{Int}, \text{Int}) \rightarrow \text{Int}))}_{:: \text{Int}}$$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

241/1700

# Beispiel 1: Konsumation komplexer Arg. (2)

->> (( (+) 120) 13)  
:: Int -> Int -> Int :: Int :: Int  
:: Int -> Int  
:: Int

->> ((120+) 13)  
:: Int -> Int :: Int  
:: Int

->> 133  
:: Int

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

242/170

# Beispiel 2: Konsumation komplexer Argumente

```
type I = Int
```

```
op :: (I -> I) -> I -> (I -> I) -> I -> (I -> I -> I) -> I
op f m g n h = h (f m) (g n)
```

Vollständig, aber nicht überflüssig geklammert:

```
op :: ((I -> I) -> (I -> ((I -> I) -> (I -> ((I -> (I -> I)) -> I))))))
op f m g n h = ((h (f m)) (g n))
```

Aufrufbeispiel:

```
op fac 5 fib 8 ggt ->>
```

```
(((op fac) 5) fib) 8) ggt)
```

```
:: (I -> ((I -> I) -> (I -> ((I -> (I -> I)) -> I))))
```

```
:: ((I -> I) -> (I -> ((I -> (I -> I)) -> I)))
```

```
:: (I -> ((I -> (I -> I)) -> I))
```

```
:: ((I -> (I -> I)) -> I)
```

```
:: I
```

```
->> ggt (fac 5) (fib 8) ->> ((ggt (fac 5)) (fib 8))
```

```
->> ((ggt 120) 21) ->> 3
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

243/170

# Klammereinsparungen für Funktionsterme (1)

...anhand einiger Beispiele:

```
fib :: Int -> Int
```

```
fib 0 = 0
```

```
fib 1 = 1
```

```
fib n = ((fib (n-2)) + (fib (n-1)))
```

Der vollständig, aber nicht überflüssig geklammerte Ausdruck

► `((fib (n-2)) + (fib (n-1)))`

kann bedeutungsgleich verkürzt werden zu:

► `fib (n-2) + fib (n-1) (*)`

...aber nicht weiter:

► `fib n-2 + fib n-1` entspricht vollständig geklammert:  
`((fib n) - 2) + ((fib n) - 1) (**)`

Somit erhalten wir:

`(*) fib (6-2) + fib (6-1) ->> fib 4 + fib 5 ->> 3 + 5 ->> 8`

`(**) fib 6-2 + fib 6-1 ->> (8-2) + (8-1) ->> 6+7 ->> 13`

# Klammereinsparungen für Funktionsterme (2)

Die vollständig, aber nicht überflüssig geklammerten Ausdrücke

- ▶ `((fac (fib 6)) - 1) ->> 119`
- ▶ `(fac ((fib 6) - 1)) ->> 24`
- ▶ `(fac (fib (6 - 1))) ->> 6`

können bedeutungsgleich verkürzt werden zu:

- ▶ `fac (fib 6) - 1 ->> fac 5 - 1 ->> 120 - 1 ->> 119`
- ▶ `fac (fib 6 - 1) ->> fac (5 - 1) ->> fac 4 ->> 24`
- ▶ `fac (fib (6 - 1)) ->> fac (fib 5) ->> fac 3 ->> 6`

Weitere Klammern können ohne Bedeutungsänderung nicht eingespart werden.

# Funktionspfeil vs. Kreuzprodukt (1)

Eine naheliegende Frage im Zshg. mit der Funktion `ersetze`:

- ▶ Warum so **viele Pfeile** (`->`), warum so **wenige Kreuze** (`×`) in der Signatur von `ersetze`?
- ▶ Warum nicht

`'ersetze :: (Txt × Vork × Alt × Neu) -> Txt'`  
statt

`ersetze :: Txt -> Vork -> Alt -> Neu -> Txt?`

**Beachte:** Das Kreuzprodukt in Haskell wird durch Tupelbeistrich ausgedrückt, d.h. `'` statt `×`. Die korrekte **Haskell-Spezifikation** für die Kreuzproduktvariante lautete daher:

`ersetze :: (Txt,Vork,Alt,Neu) -> Txt`

# Funktionspfeil vs. Kreuzprodukt (2)

## Beide Formen

- ▶ sind möglich, sinnvoll und berechtigt.

## Funktionspfeil

- ▶ führt jedoch zu höherer (Anwendungs-) Flexibilität als Kreuzprodukt, da partielle Auswertung von Funktionen möglich ist.
- ▶ ist daher in funktionaler Programmierung die weitaus häufiger verwendete Form.

## Zur Illustration:

- ▶ Berechnung der Binomialkoeffizienten.

# Funktionspfeil vs. Kreuzprodukt (3)

Vergleiche die Funktionspfeilform:

```
binom :: Integer -> Integer -> Integer
binom n k
  | k==0 || n==k = 1
  | otherwise    = binom (n-1) (k-1) + binom (n-1) k
```

...mit der Kreuzproduktform:

```
binom' :: (Integer,Integer) -> Integer
binom' (n,k)
  | k==0 || n==k = 1
  | otherwise    = binom' (n-1,k-1) + binom' (n-1,k)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

248/1700



# Funktionspfeil vs. Kreuzprodukt (4)

Die höhere Flexibilität der Funktionspfeilform zeigt sich in der Anwendungssituation:

Der Funktionsterm `(binom 45)`

- ▶ ist von funktionalem Typ `(Integer -> Integer)`, eine Funktion, die ganze Zahlen in sich abbildet.
- ▶ liefert angewendet auf eine natürliche Zahl  $k$  die Anzahl der Möglichkeiten, auf die man  $k$  Elemente aus einer 45-elementigen Grundgesamtheit herausgreifen kann:  
`((binom 45)` entspricht der Funktion `k_aus_45)`

# Funktionspfeil vs. Kreuzprodukt (5)

Wir können den Funktionsterm `(binom 45)` deshalb auch benutzen, um *in argumentfreier Weise* eine neue Funktion zu definieren, z.B. die Funktion `k_aus_45` (vgl. Kap. 1.1.1):

```
k_aus_45 :: Integer -> Integer
k_aus_45 = binom 45 -- arg.frei: k_aus_45 ist nicht
                    -- von einem Arg. gefolgt
```

Die Funktion `k_aus_45` und der Funktionsterm `(binom 45)` bezeichnen *dieselbe Funktion*; sie sind Synonyme.

Aufrufe folgender Form sind deshalb möglich:

```
(binom 45) 6 ->> 8.145.060
binom 45 6   ->> 8.145.060 -- Klammereinsparungsrr.
k_aus_45 6   ->> binom 45 6 ->> 8.145.060
```

# Funktionspfeil vs. Kreuzprodukt (6)

Beachte: Auch die Funktion

```
binom' :: (Integer,Integer) -> Integer
```

ist im Haskell-Sinn einstellig.

Folgende Schreibweise macht dies besonders deutlich:

```
type IntPair = (Integer,Integer)
```

```
binom' :: IntPair -> Integer -- 1 Argument: 1-stellig
```

```
binom' p
```

```
  | snd(p) == 0 || fst(p)==snd(p) = 1
```

```
  | otherwise = binom' (fst(p)-1,snd(p)-1)  
                + binom' (fst(p)-1,snd(p))
```

`p` vom Typ `IntPair`, das eine Argument von `binom'` ist von einem `Paartyp`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

251/1700

# Funktionspfeil vs. Kreuzprodukt (7)

Beachte: `binom'` bietet nicht die Flexibilität von `binom`:

- ▶ `binom'` konsumiert ihr `eines` Argument `p` vom Paartyp `(Integer,Integer)` und liefert unmittelbar ein Resultat vom elementaren Typ `Integer`.

```
binom' (45,6) ->> 8.145.060 :: Integer
```

- ▶ ein funktionales Zwischenresultat entsteht anders als bei `binom` nicht.

- ▶ Eine lediglich `teilweise Versorgung mit Argumenten` und damit `partielle Auswertung` von `binom'` ist `nicht möglich`.

Aufrufe der Form:

```
binom' 45
```

sind `syntaktisch inkorrekt` und führen zu Fehlermeldungen.

# Vordef. arithmetische Operationen in Pfeilform

Auch die arithmetischen (und viele weitere) Operationen sind in Haskell aus diesem Grund in der Funktionspfeilform vordefiniert:

```
(+) :: Num a => a -> a -> a  
(* ) :: Num a => a -> a -> a  
(- ) :: Num a => a -> a -> a  
...
```

Nachstehend instantiiert für den Typ `Int`:

```
(+) :: Int -> Int -> Int  
(* ) :: Int -> Int -> Int  
(- ) :: Int -> Int -> Int  
...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

253/170

# Funktionsstelligkeiten: Mathematik vs. Haskell

...unterschiedliche Sichtweisen und Akzentsetzungen.

**Mathematik:** Betonung der 'Teile' – eine Funktion der Form:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

wird zweistellig angesehen:  $(\cdot) : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

*Allgemein:*  $f : M_1 \times \dots \times M_n \rightarrow M$  hat Stelligkeit  $n$ .

**Haskell:** Betonung des 'Ganzen' – eine Funktion der Form:

```
type I = Integer
```

```
binom' (n,k)
```

```
  | k==0 || n==k = 1
```

```
  | otherwise = binom' (n-1,k-1) + binom' (n-1,k)
```

wird einstellig angesehen:  $\text{binom}' :: (I,I) \rightarrow I$

*Allgemein:*  $f :: (M_1, \dots, M_n) \rightarrow M$  hat Stelligkeit  $1$ .

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

254/170

# Zusammenfassung (1)

Für **Haskell** gilt:

Die Klammerung **unvollständig geklammerter**

- ▶ **Funktionssignaturen** ist **rechtsassoziativ**
- ▶ **Funktionsterme** ist **linksassoziativ**

zu vervollständigen.

**Funktionen** sind

- ▶ **einstellig**; sie konsumieren stets **ein** Argument zur Zeit.

Argumente und Werte von **Funktionen** und **Funktionstermen**

- ▶ können **elementaren**, **zusammengesetzten** oder **funktionalen Typs** sein.

# Zusammenfassung (2)

## Klammern in Signaturen und Funktionstermen

- sind mehr als schmückendes Beiwerk; sie bestimmen die Bedeutung.

Wann immer eine von den Klammereinsparungsregeln induzierte abweichende Argument- oder/und Resultatstruktur gewollt ist, muss dies durch

- explizite Klammerung in Signatur und Funktionsterm ausgedrückt werden.



# Zusammenfassung (3)

...exakte, saloppe Sprechweise für curryfizierte Funktionen.

Exakt: `binom` ist eine 1-stellige Funktion:

```
binom :: Integer -> (Integer -> Integer)
(binom n) k = if ... then ... else ...
:: (Integer -> Integer)
```

...Sichtweise von `binom` als 1-stellige Funktion, die ganze Zahlen auf 1-stellige Funktionen abbildet, die ganze Zahlen auf ganze Zahlen abbilden.

Salopp: `binom` ist eine 2-stellige Funktion:

```
binom :: Integer -> Integer -> Integer
binom n k = if ... then ... else ...
:: Integer
```

...Sichtweise von `binom` als 2-stellige Funktion, die zwei ganze Zahlen auf ganze Zahlen abbildet.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV  
257/170

# Übungsaufgabe 3.2.1

1. Wie liest sich die **Signatur** der Editor-Funktion

`ersetze :: Txt -> Vork -> Alt -> Neu -> Txt`  
gemäß

- 1.1 **exakter** Sichtweise:

`ersetze :: Txt -> (Vork -> (Alt -> (Neu -> Txt)))`

- 1.2 **salopper** Sichtweise:

`ersetze :: Txt -> Vork -> Alt -> Neu -> Txt`

2. Wie sind die **Signaturen** der Funktionen

`f :: (I -> I -> I) -> I -> [I] -> I`

`g :: (I -> I) -> (I -> I) -> [I] -> I`

`h :: I -> (I -> I -> I) -> [I] -> I`

**exakt** bzw. **salopp** zu lesen (type I = Int)?

3. Wiederholen Sie die Übung mit anderen curryfizierten Funktionssignaturen aus den bisherigen Kapiteln oder/und dem Standard-Präludium.

# Kapitel 3.3

## Curryfizierte, uncurryfizierte Funktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

**3.3**

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Scharf oder mild

...curryfiziert oder uncurryfiziert, das ist hier die Frage.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Curryfiziert und uncurryfiziert

...bezeichnen bestimmte ineinander überführbare Deklarationsweisen für Funktionen.

Entscheidend für die Unterscheidung ist die

- ▶ Art der Konsumation der Argumente.

Erfolgt die Konsumation

- ▶ Einzelne Argumente für Argumente: **curryfiziert**
- ▶ Alle auf einmal als Tupel: **uncurryfiziert**

Implizit liefert dies eine Unterscheidung in

- ▶ **curryfizierte** Funktionen
- ▶ **uncurryfizierte** Funktionen

# Curryfiziert vs. uncurryfiziert deklariert (1)

...anhand eines Beispiels:

► `binom :: Integer -> Integer -> Integer`

...ist **curryfiziert** deklariert.

► `binom' :: (Integer,Integer) -> Integer`

...ist **uncurryfiziert** deklariert.

## Curryfiziert vs. uncurryfiziert deklariert (2)

Curryfiziert deklariertes `binom`:

```
binom :: Integer -> Integer -> Integer
```

```
binom n k
```

```
  | k==0 || n==k = 1
```

```
  | otherwise = binom (n-1) (k-1) + binom (n-1) k
```

```
binom 45 6 ->> (binom 45) 6 ->> 8.145.060
```

$\underbrace{\quad\quad\quad}_{:: \text{Integer} \rightarrow \text{Integer}}$   
 $\underbrace{\quad\quad\quad}_{:: \text{Integer}}$

Uncurryfiziert deklariertes `binom'`:

```
binom' :: (Integer,Integer) -> Integer
```

```
binom' (n,k)
```

```
  | k==0 || n==k = 1
```

```
  | otherwise = binom' (n-1,k-1) + binom' (n-1,k)
```

```
binom' (45,6) ->> 8.145.060
```

$\underbrace{\quad\quad\quad}_{:: \text{Integer}}$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

263/170

# Informell

Curryfizieren ersetzt

- ▶ Produkt-/Tupelbildung ' $\times$ ' durch Funktionspfeil ' $\rightarrow$ '.

Uncurryfizieren ersetzt

- ▶ Funktionspfeil ' $\rightarrow$ ' durch Produkt-/Tupelbildung ' $\times$ '.

**Bemerkung:** Die Bezeichnung erinnert an [Haskell B. Curry](#); die Idee selbst ist weit älter und geht auf [Moses Schönfinkel](#) aus der [Mitte der 1920er-Jahre](#) zurück.



# Die Funktionale **curry** und **uncurry**

..als Mittler zwischen **curryfizzierter** und **uncurryfizzierter** Darstellung:

Das Funktional **curry**:

<b>curry</b> :: $\underbrace{((a,b) \rightarrow c)}_{\text{Argumenttyp von curry: uncurryfiziert!}}$	$\rightarrow$	$\underbrace{(a \rightarrow b \rightarrow c)}_{\text{Resultattyp von curry: curryfiziert!}}$
--	---------------	--

Das Funktional **uncurry**:

<b>uncurry</b> :: $\underbrace{(a \rightarrow b \rightarrow c)}_{\text{Argumenttyp von uncurry: curryfiziert!}}$	$\rightarrow$	$\underbrace{((a,b) \rightarrow c)}_{\text{Resultattyp von uncurry: uncurryfiziert!}}$
--	---------------	--

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

265/170

# Die Implementierungen v. **curry** u. **uncurry** (1)

Das Funktional **curry**:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)    -- x, y wird zu (x,y)
                        -- zusammengesetzt und so
                        -- für f verarbeitbar
```

Das Funktional **uncurry**:

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry g (x,y) = g x y  -- (x,y) wird in x, y
                        -- getrennt und so
                        -- für g verarbeitbar
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

266/1700

# Die Implementierungen v. **curry** u. **uncurry** (2)

...in größerem Detail:

Das Funktional **curry**:

$$\begin{array}{l} \text{curry} :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c) \\ \text{curry } f \quad \quad \quad x \quad y = f \ (x,y) \\ \underbrace{\quad \quad \quad}_{:: (a,b) \rightarrow c} \quad \underbrace{\quad}_{:: a} \quad \underbrace{\quad}_{:: b} \quad \underbrace{\quad \quad \quad}_{\substack{:: (a,b) \\ :: c}} \end{array}$$

Das Funktional **uncurry**:

$$\begin{array}{l} \text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow ((a,b) \rightarrow c) \\ \text{uncurry } g \quad \quad \quad (x,y) = g \quad \quad \quad x \quad y \\ \underbrace{\quad \quad \quad}_{:: (a \rightarrow b \rightarrow c)} \quad \underbrace{\quad \quad \quad}_{:: (a,b)} \quad \underbrace{\quad \quad \quad}_{\substack{:: a \quad :: b \\ :: c}} \end{array}$$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

267/170

# Zur Klammerung von **curry** und **uncurry**

...vollständig, aber nicht überflüssig geklammert:

**curry** :: ((a,b) -> c) -> (a -> (b -> c))

**curry** f x y = f (x,y)

**uncurry** :: ((a -> (b -> c)) -> ((a,b) -> c))

**uncurry** g (x,y) = g x y

...minimal geklammert gemäß Klammereinsparungsregeln:

**curry** :: ((a,b) -> c) -> a -> b -> c

**curry** f x y = f (x,y)

**uncurry** :: (a -> b -> c) -> (a,b) -> c

**uncurry** g (x,y) = g x y

# curry: Schritt für Schritt zur Definition

```
curry :: ((a,b) -> c) -> (a -> (b -> c))  
curry f x y = f (x,y)
```

Sei  $f$  eine Funktion mit Signatur

```
f :: ((a,b) -> c)
```

Mit  $f$  erhalten wir für die Signaturen der Funktionsterme:

```
curry :: ((a,b) -> c) -> (a -> (b -> c))  
(curry f) :: (a -> (b -> c))  
((curry f) x) :: (b -> c)  
(((curry f) x) y) :: c
```

Entsprechend erhalten wir für den Typ des rechtss. Fkt-Terms:

```
((curry f) x) y = f (x,y) :: c
```

Nach Einsparung von Klammern erhalten wir insgesamt:

```
curry :: ((a,b) -> c) -> (a -> (b -> c))  
curry f x y = f (x,y)
```

# Anwendung von `curry`

Sei `f` uncurryfiziert gegebene Funktion mit Signatur

`f :: ((a,b) -> c)`

Definiere

`g :: (a -> (b -> c))`

`g = curry f` `-- argumentfrei!`

Damit

`f (x,y) = g x y = (curry f) x y = curry f x y`

# Übungsaufgabe 3.3.1

Vollziehe die **Schritt-für-Schritt-Entwicklung** zur Definition und Anwendung von **curry** für **uncurry** nach, d.h. entwickle nach dem Beispiel von **curry** Schritt für Schritt die Definition und Anwendung von **uncurry**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

**3.3**

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

271/170

# Die Funktionale **curry** und **uncurry**

...bilden

- **uncurryfizierte** Funktionen auf ihre **curryfizierten** Gegenstücke ab:

Für **uncurryfiziertes**  $f :: (a,b) \rightarrow c$  ist

**curry**  $f :: a \rightarrow (b \rightarrow c)$

**curryfiziert** (entsprechend  $g :: a \rightarrow (b \rightarrow c)$ ).

- **curryfizierte** Funktionen auf ihre **uncurryfizierten** Gegenstücke ab:

Für **curryfiziertes**  $g :: a \rightarrow (b \rightarrow c)$  ist

**uncurry**  $g :: (a,b) \rightarrow c$

**dec Curryfiziert** (entsprechend  $f :: (a,b) \rightarrow c$ ).



# Anwendungen von `curry` und `uncurry`

Betrachte:

```
binom :: Integer -> Integer -> Integer
```

```
binom' :: (Integer,Integer) -> Integer
```

und

```
curry    :: ((a,b) -> c) -> (a -> b -> c)
```

```
uncurry  :: (a -> b -> c) -> ((a,b) -> c)
```

Anwendung von `curry` und `uncurry` liefert:

```
curry binom' :: Integer -> Integer -> Integer
```

```
uncurry binom :: (Integer,Integer) -> Integer
```

Somit sind folgende Aufrufe möglich und gültig:

```
curry binom' 45 6    ->> binom' (45,6) ->> 8.145.060
```

```
uncurry binom (45,6) ->> binom 45 6    ->> 8.145.060
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

273/170

# Curryfiziert oder uncurryfiziert?

...das ist die Frage.

Geschmackssache? Notationelle Spielerei?

- ▶  $f\ x, f\ x\ y, f\ x\ y\ z, \dots$  vs.  $f(x), f(x,y), f(x,y,z), \dots$   
Allenfalls bei oberflächlicher Betrachtung.

Denn es gilt: Nur curryfizierte Funktionen unterstützen das

- ▶ Prinzip partieller Auswertung und damit das Prinzip:  
     $\rightsquigarrow$  Funktionen liefern Funktionen als Ergebnis!

Beispiel: Die für das Argument 45 partiell ausgewertete Funktion `binom` liefert als Resultat eine einstellige Funktion, die Funktion `k_aus_45 :: Integer -> Integer` definiert durch `k_aus_45 = (binom 45)`.

Die Bevorzugung curryfizierter Formen ist deshalb sachlich gut begründet, vorteilhaft und in der Praxis vorherrschend.

# Faustregel für Funktionsdefinitionen

...curryfiziert, wo möglich, uncurryfiziert, wo nötig.

(vgl. die Funktionen `binom` und `binom'` vom Anfang dieses Abschnitts.)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

275/1700

# Kapitel 3.4

## Operatoren, Präfix- und Infixverwendung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

**3.4**

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Operatorverwendung

## Präfixverwendung

- ▶ den Operanden vorangestellt:

*Beispiele:* `fac 5`, `binom (45,6)`, `reverse "desserts"`,  
`quickSort [4,2,1,9,3,7,5]`,...

## Infixverwendung

- ▶ zwischen die Operanden gestellt:

*Beispiele:* `2 + 3`, `5 * 7`, `5 ^ 3`, `4 : [3,2,1]`, `[1,2,3,4] !! 2`,  
`[3,2,1] ++ [1,2,3]`,...

## Postfixverwendung

- ▶ den Operanden nachgestellt:

*Beispiele:* In Haskell keine; in der Mathematik wenige, etwa die Fakultätsfunktion “!”; regelmäßig bei Verwendung “umgekehrt polnischer Notation”.

# Operatorverwendung in Haskell

## Präfixverwendung ist

- ▶ Regelfall, insbesondere für alle selbstdeklarierten Operatoren (d.h. selbstdeklarierte Funktionen).

### Beispiele:

- ▶ Vordefinierte Funktionen: `div`, `reverse`, `zip`,...
- ▶ Selbstdefinierte Funktionen: `fac`, `binom`, `quicksort`,...

## Infixverwendung ist

- ▶ Regelfall für einige vordefinierte Operatoren und Relatoren, darunter viele arithmetische Operatoren und Relatoren.

*Beispiele:* `2 + 3`, `5 * 7`, `5 ^ 3`, `4 : [3,2,1]`, `[1,2,3,4] !! 2`,  
`[3,2,1] ++ [1,2,3]`, `[3,2,1] == [1,2,3]`,  
`4 <= 5`, `"Fun" < "More Fun"`,...

# Erweiterte Verwendungsmöglichkeiten

...gelten für **binäre Operatoren** in Haskell.

**Infix- und Präfixverwendung** ist möglich für

- ▶ **vordefinierte und selbstdefinierte Binäroperatoren.**

**Allgemein:** Wird der Binäroperator **bop** im Regelfall als

- ▶ **Präfixoperator** verwendet, so kann **bop** mit Hochkommata als **Infixoperator** **'bop'** verwendet werden.

*Beispiele:* 45 **'binom'** 6, 3 **'mult'** 5

(statt standardmäßig: **binom** 45 6, **mult** 3 5)

- ▶ **Infixoperator** verwendet, so kann **bop** geklammert als **Präfixoperator** (**bop**) verwendet werden.

*Beispiele:* **(+)** 2 3, **(++)** [3,2,1] [1,2,3]

(statt standardmäßig: 2 **+** 3, [2,1] **++** [1,2])

# Beispiel

...berechne das **Maximum dreier ganzer Zahlen**:

```
max :: Int -> Int -> Int -> Int
```

```
max p q r
```

```
  | (mx p q == p) && (p 'mx' r == p) = p
```

```
  | (mx p q == q) && (q 'mx' r == q) = q
```

```
  | otherwise                        = r
```

```
where mx :: Int -> Int -> Int
```

```
      mx p q
```

```
      | p >= q      = p
```

```
      | otherwise = q
```

**Beachte:** Der Binäroperator `mx` wird in `max` als **Präfixoperator** `(mx p q)` und **Infixoperator** `(p 'mx' r)` verwendet.



# Weitere Beispiele

...für Infix- und Präfixverwendung von Binäroperatoren anhand einiger arithmetischer Funktionen:

- ▶ Inkrement
- ▶ Dekrement
- ▶ Halbieren
- ▶ Verdoppeln
- ▶ 10er-Inkrement
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

**3.4**

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

281/170

# Infixverwendete Binäroperatoren

## ► Inkrement

```
inc :: Integer -> Integer
```

```
inc n = n + 1
```

## ► Dekrement

```
dec :: Integer -> Integer
```

```
dec n = n - 1
```

## ► Halbieren

```
hlv :: Integer -> Integer
```

```
hlv n = n `div` 2    -- Nichtstandardverwendung
```

## ► Verdoppeln

```
dbl :: Integer -> Integer
```

```
dbl n = 2 * n
```

## ► 10er-Inkrement

```
inc10 :: Integer -> Integer
```

```
inc10 n = n + 10
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

282/170

# Präfixverwendete Binäroperatoren

## ► Inkrement

```
inc :: Integer -> Integer  
inc n = (+) n 1           -- Nichtstandardverw.
```

## ► Dekrement

```
dec :: Integer -> Integer  
dec n = (-) n 1           -- Nichtstandardverw.
```

## ► Halbieren

```
hlv :: Integer -> Integer  
hlv n = div n 2           -- Standardverw.
```

## ► Verdoppeln

```
dbl :: Integer -> Integer  
dbl n = (*) 2 n           -- Nichtstandardverw.
```

## ► 10er-Inkrement

```
inc10 :: Integer -> Integer  
inc10 n = (+) n 10        -- Nichtstandardverw.
```

# Punktfrei als partiell ausgewertete Funktionen

## ► Inkrement

```
inc :: Integer -> Integer
inc = (+) 1
```

## ► Eins\_minus (nicht Dekrement)

```
eins_minus :: Integer -> Integer
eins_minus = (-) 1      -- (-) nicht kommutativ
```

## ► Zwei\_durch (nicht Halbieren)

```
zwei_durch :: Integer -> Integer
zwei_durch = div 2      -- div nicht kommutativ
```

## ► Verdoppeln

```
dbl :: Integer -> Integer
dbl = (*) 2
```

## ► 10er-Inkrement

```
inc10 :: Integer -> Integer
inc10 = (+) 10
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

284/170

# Operandenstellung und Klammerung

...führen uns zu sog. **Operatorabschnitten**:

## ► Inkrement

```
inc :: Integer -> Integer
```

```
inc = (+1)
```

## ► Eins\_minus

```
eins_minus :: Integer -> Integer
```

```
eins_minus = (1-)
```

## ► Verdoppeln

```
dbl :: Integer -> Integer
```

```
dbl = (2*)
```

## ► Halbieren

```
hlv :: Integer -> Integer
```

```
hlv = ('div' 2)
```

**Beachte** die unterschiedliche Klammerung und Operandenstellung in `inc` und `eins_minus` sowie in `dbl` und `hlv`.

# Kapitel 3.5

## Operatorabschnitte

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

**3.5**

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Operatorabschnitte (1)

Partiell ausgewertete Binäroperatoren heißen in Haskell

- ▶ Operatorabschnitte (engl. *operator sections*)

Beispiele:

- ▶  $(*2)$  `dbl`, die Funktion, die ihr Argument verdoppelt  
 $(\lambda x. x * 2)$
- ▶  $(2*)$  `dbl`, s.o.  $(\lambda x. 2 * x)$
- ▶  $(2<)$  `2_kleiner_als_x`, das Prädikat, das überprüft, ob sein Argument größer als 2 ist  $(\lambda x. 2 < x)$
- ▶  $(<2)$  `x_kleiner_als_2`, das Prädikat, das überprüft, ob sein Argument kleiner als 2 ist  $(\lambda x. x < 2)$
- ▶  $(2:)$  `headAppend`, die Funktion, die 2 an den Anfang einer typkompatiblen Liste setzt  $(\lambda xs. (2 : xs))$
- ▶ ...

# Operatorabschnitte (2)

Beispiele (fgs.):

- ▶ `(+1)`, `(1+)` `inc`, die Funktion, die ihr Argument um 1 erhöht  $(\lambda x. x + 1)$  bzw.  $(\lambda x. 1 + x)$
- ▶ `(1-)` `eins_minus`, die Funktion, die ihr Argument von 1 abzieht  $(\lambda x. 1 - x)$
- ▶ `(-1)` kein Operatorabschn., sondern d. Zahl '-1'.
- ▶ `(+(-1))` `dec`, die Funktion, die ihr Argument um 1 erniedrigt  $(\lambda x. x + (-1))$  bzw.  $(\lambda x. x - 1)$
- ▶ `('div' 2)` `hlv`, die Funktion, die ihr Argument ganzzahlig halbiert  $(\lambda x. x \text{ div } 2)$
- ▶ `(2 'div')` `zwei_durch`, die Funktion, die 2 ganzzahlig durch ihr Argument teilt  $(\lambda x. 2 \text{ div } x)$
- ▶ ...
- ▶ `(div 2)`,  
`div 2` `zwei_durch`, s.o.  $(\lambda x. 2 \text{ div } x)$ ; keine echten Operatorabschnitte, sondern gewöhnliche Präfixoperatorverwendung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

288/1700



# Operatorabschnitte (3)

Operatorabschnitte können in Haskell gebildet werden mit

- ▶ vordefinierten und selbstdefinierten binären Operatoren.

Beispiele für die curryfizierte Funktion `binom` (vgl. Kap. 3.2):

- ▶ `(binom 45)`      `45_über_k`, die Funktion `k_aus_45`.
- ▶ `(45 'binom')`    `45_über_k`, s.o.
- ▶ `('binom' 6)`     `n_über_6`, die Funktion `6_aus_n`.
- ▶ ...

**Beachte:** Mit der uncurryfzierten Funktion `binom'` (vgl. Kapitel 3.2) können keine Operatorabschnitte gebildet werden.

# Anwendung: Punktfreie, argumentlose

...Funktionsdefinitionen mit Operatorabschnitten:

## ► 45\_über\_k bzw. k\_aus\_45

```
k_aus_45 :: Integer -> Integer
```

```
k_aus_45 = binom 45
```

```
k_aus_45 :: Integer -> Integer
```

```
k_aus_45 = (45 'binom')
```

## ► n\_über\_6 bzw. 6\_aus\_n

```
sechs_aus_n :: Integer -> Integer
```

```
sechs_aus_n = ('binom' 6)
```

## ► Inkrement

```
inc :: Integer -> Integer
```

```
inc = (+1)
```

## ► Verdoppeln

```
dbl :: Integer -> Integer
```

```
dbl = (2*)
```

## ► ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

290/170

# Nichtkommutative Operatoren

...benötigen Obacht bei der Bildung von Operatorabschnitten.

Infix- und Präfixbenutzung hat für nichtkommutative Operatoren einen Bedeutungsunterschied. Am Beispiel von `div`:

- ▶ Infixverwendung führt zu den Funktionen `hlv` und `zwei_durch`.
- ▶ Präfixverwendung führt zur Funktion `zwei_durch`.

bei ansonsten gleicher partieller Auswertung.

# Am Beispiel von div für hlv und zwei\_durch

## ► Halbieren (durch\_zwei) (div infixverwendet)

```
hlv :: Integer -> Integer
hlv = ('div' 2)           -- Operatorabschnitt

hlv 5 ->> 2, hlv 10 ->> 5, hlv 15 ->> 7
```

## ► zwei\_durch (div präfixverwendet)

```
zwei_durch :: Integer -> Integer
zwei_durch = div 2       -- Präfixverwendung
-- bedeutungsgleich mit:
zwei_durch = (2 'div')   -- Operatorabschnitt

zwei_durch 5 ->> 0, zwei_durch 2 ->> 1,
zwei_durch 1 ->> 2
```

# Am Beispiel von `(-)` für `eins_minus`

## ► `eins_minus ((-))` infixverwendet

```
eins_minus :: Integer -> Integer
```

```
eins_minus = (1-)          -- Operatorabschnitt
```

```
eins_minus 5    ->> -4, eins_minus 1 ->> 0,
```

```
eins_minus (-1) ->> 2
```

## ► `eins_minus ((-))` präfixverwendet

```
eins_minus :: Integer -> Integer
```

```
eins_minus = (-) 1        -- Präfixverwendung
```

```
-- bedeutungsgleich mit:
```

```
eins_minus = (1-)        -- Operatorabschnitt
```

Beachte: `(-1)` repräsentiert die Zahl `'-1'`, keinen Operatorabschnitt.

# Am Beispiel von (+) für inc

...kein Unterschied wg. Kommutativität der Addition:

## ► Inkrement ((+) infixverwendet)

```
inc :: Integer -> Integer
```

```
inc = (+1)                                -- Operatorabschnitt
```

```
-- bedeutungsgleich mit:
```

```
inc = (1+)                                -- Operatorabschnitt
```

```
inc 5 ->> 6, inc 10 ->> 11, inc 15 ->> 16
```

## ► Inkrement ((+) präfixverwendet)

```
inc :: Integer -> Integer
```

```
inc = (+) 1                               -- Präfixverwendung
```

```
-- entspricht:
```

```
inc = (1+)
```

```
inc 5 ->> 6, inc 10 ->> 11, inc 15 ->> 16
```

# Am Beispiel von (\*) für dbl

...kein Unterschied wg. Kommutativität der Multiplikation:

## ► Verdoppeln (\*) infixverwendet)

```
dbl :: Integer -> Integer
```

```
dbl = (*2) -- Operatorabschnitt
```

```
-- bedeutungsgleich mit:
```

```
dbl = (2*) -- Operatorabschnitt
```

```
dbl 5 ->> 10, dbl 10 ->> 20, dbl 15 ->> 30
```

## ► Verdoppeln (\*) präfixverwendet)

```
dbl :: Integer -> Integer
```

```
dbl = (*) 2 -- Präfixverwendung
```

```
-- entspricht:
```

```
dbl = (2*)
```

```
dbl 5 ->> 10, dbl 10 ->> 20, dbl 15 ->> 30
```

# Zusammenfassung

...**Operatorabschnitte**: Notationelle Abkürzungen ('syntaktischer Zucker') für **anonyme  $\lambda$ -Abstraktionen**.

**Im Detail**: Ist **op** ein Binäroperator und sind **x** und **y** typgeeignete Operanden für **op**, dann heißen die Ausdrücke:

►  $(op)$ ,  $(x \ op)$ ,  $(op \ y)$

**Operatorabschnitte**, die für folgende Funktionen stehen:

►  $(op) = (\lambda x. (\lambda y. x \ op \ y))$

►  $(x \ op) = (\lambda y. x \ op \ y)$

►  $(op \ y) = (\lambda x. x \ op \ y)$

und somit besonders knappe Funktionsdefinitionen erlauben (Sonderfall: Der Subtraktionsoperator **(-)**).



# Kapitel 3.6

## Angemessene, unangemessene Funktionsdefinitionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

**3.6**

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Total vs. partiell definierte Funktionen (1)

Total definierte Funktionen sind die Ausnahme, partiell definierte Funktionen die Regel.

Betrachte z.B. die Funktionen `fac` und `fib`, deren Auswertung nur für nichtnegative Argumente terminiert und definiert ist:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)

fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

298/1700

# Total vs. partiell definierte Funktionen (2)

...als Implementierungen der (auf den natürlichen Zahlen total definierten) **Fakultäts- und Fibonacci-Funktion**:

$$! : \mathbb{IN} \rightarrow \mathbb{IN}$$

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n - 1)! & \text{sonst} \end{cases}$$

$$fib : \mathbb{IN} \rightarrow \mathbb{IN}$$

$$fib(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ fib(n - 2) + fib(n - 1) & \text{sonst} \end{cases}$$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Transparenz der Partialität

Partialität der Implementierungen von `fac` und `fib`

- i.w. **technisch induziert** (Abwesenheit eines Datentyps für natürliche Zahlen).

Explizite, transparente Sichtbarmachung der Partialität ist dennoch **sinnvoll, angemessen** und auch **einfach möglich**:

```
fac :: Int -> Int
fac n | n == 0 = 1
      | n >= 1 = n * fac (n - 1)
      | otherwise = error "undefiniert"
```

```
fib :: Int -> Int
fib n | n == 0 = 1
      | n == 1 = 1
      | n >= 2 = fib (n-2) + fib (n-1)
      | otherwise = error "undefiniert"
```

# Partiell definierte Funktionen

...sind auch  $f$ ,  $g$  und  $h$ :

$$f : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$f(z) = \begin{cases} 2 & \text{falls } z \geq 1 \\ \text{undef} & \text{sonst} \end{cases}$$

$$g : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$g(z) = \begin{cases} 2^z & \text{falls } z \geq 1 \\ \text{undef} & \text{sonst} \end{cases}$$

$$h : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$h(z) = \begin{cases} 2^1 & \text{falls } z = 1 \\ 2^{|z|+2} & \text{falls } z \leq 0 \\ \text{undef} & \text{sonst} \end{cases}$$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

301/170

# Angemessene Implementierungen von f, g, h

...die Partialität der Implementierungen von f, g und h liegt transparent und offen zutage:

```
f :: Integer -> Integer
f z | z >= 1      = 2
    | otherwise = error "undefiniert"
```

```
g :: Integer -> Integer
g z | z >= 1      = 2^z
    | otherwise = error "undefiniert"
```

```
h :: Integer -> Integer
h z | z == 1      = 2
    | z <= 0      = 2^((abs z)+2)
    | otherwise = error "undefiniert"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

302/1700

# Unangemessene Implementierung von $f$

Betrachte folgende **intransparente Implementierung** von  $f$ :

```
f :: Integer -> Integer
f 1 = 2
f x = 2 * (f x)
```

**Verschleiernd** und **intransparent**, auch wenn man sich durch Nachrechnen darüber versichern kann:

Die **Auswertung** von  $f$  terminiert für  $n = 1$ :

```
f 1 ->> 2
```

...aber **für keinen** von 1 verschiedenen Argumentwert.

# Auswertungsbeispiele für f

Die **Auswertung** von **f** terminiert für  $n = 1$ :

f 1       $\rightarrow$  2

Die **Auswertung** von **f** terminiert nicht für  $n \neq 1$ :

f (-9)  $\rightarrow$  2 \* (f (-9))  $\rightarrow$  2 \* (2 \* (f (-9)))  
 $\rightarrow$  2 \* (2 \* (2 \* (f (-9))))  $\rightarrow$  ...

f (-1)  $\rightarrow$  2 \* (f (-1))  $\rightarrow$  2 \* (2 \* (f (-1)))  
 $\rightarrow$  2 \* (2 \* (2 \* (f (-1))))  $\rightarrow$  ...

f 0       $\rightarrow$  2 \* (f 0)       $\rightarrow$  2 \* (2 \* (f 0))  
 $\rightarrow$  2 \* (2 \* (2 \* (f 0)))       $\rightarrow$  ...

f 2       $\rightarrow$  2 \* (f 2)       $\rightarrow$  2 \* (2 \* (f 2))  
 $\rightarrow$  2 \* (2 \* (2 \* (f 2)))       $\rightarrow$  ...

f 3       $\rightarrow$  2 \* (f 3)       $\rightarrow$  2 \* (2 \* (f 3))  
 $\rightarrow$  2 \* (2 \* (2 \* (f 3)))       $\rightarrow$  ...

f 9       $\rightarrow$  2 \* (f 9)       $\rightarrow$  2 \* (2 \* (f 9))  
 $\rightarrow$  2 \* (2 \* (2 \* (f 9)))       $\rightarrow$  ...



# Angemessenheit vs. Unangemessenheit für $f$

...beide Implementierungen der Funktion  $f$  sind formal und inhaltlich korrekt, dennoch:

Die Deklaration:

```
f :: Integer -> Integer
f 1 = 2
f x = 2 * (f x)
```

...ist unangemessen, weil intransparent und verschleiernd.

```
f :: Integer -> Integer
f z | z = 1      = 2
    | otherwise = error "undefiniert"
```

...ist angemessen, weil transparent und offen legend.

# Unangemessene Implementierungen von $g$ , $h$

Betrachte folgende **intransparente Implementierungen** von  $g$  und  $h$ :

```
g :: Integer -> Integer
```

```
g 1      = 2
```

```
g (x+1) = 2 * (g x)
```

```
h :: Integer -> Integer
```

```
h 1 = 2
```

```
h x = 2 * (h (x+1))
```

**Bemerkung:** Muster der Form  $(x+1)$  wie in der Definition von  $g$  nicht mehr zulässig in neueren Haskell-Versionen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

306/170

# Auswertungsbeispiele für g

Die **Ausw.** von **g** terminiert für **echt positive** Argumentwerte:

$g\ 1 \rightarrow 2$

$g\ 2 \rightarrow g\ (1+1) \rightarrow 2 * (g\ 1) \rightarrow 2 * 2 \rightarrow 4$

$g\ 3 \rightarrow g\ (2+1) \rightarrow 2 * (g\ 2) \rightarrow 2 * g\ (1+1)$   
 $\rightarrow 2 * (2 * (g\ 1)) \rightarrow 2 * (2 * 2) \rightarrow 2 * 4$   
 $\rightarrow 8$

$g\ 9 \rightarrow g\ (8+1) \rightarrow 2 * (2 * (g\ 8)) \rightarrow \dots \rightarrow 512$

Die **Auswertung** von **g** terminiert **nicht** sonst:

$g\ 0 \rightarrow g\ ((-1)+1) \rightarrow 2 * (g\ (-1))$   
 $\rightarrow 2 * (g\ ((-2)+1)) \rightarrow 2 * (2 * (g\ (-2)))$   
 $\rightarrow 2 * (2 * (g\ ((-3)+1)))$   
 $\rightarrow 2 * (2 * (2 * (g\ (-3)))) \rightarrow \dots$

$g\ (-1) \rightarrow g\ ((-2)+1) \rightarrow 2 * (g\ (-2)) \rightarrow \dots$

$g\ (-9) \rightarrow g\ ((-10)+1) \rightarrow 2 * (g\ (-10)) \rightarrow \dots$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

307/170

# Auswertungsbeispiele für h

Die **Auswertung** von **h** terminiert für Argumentwerte  $\leq 1$ :

h 1 ->> 2

h 0 ->> 2 \* (h (0+1)) ->> 2 \* (h 1) ->> 2 \* 2  
->> 4

h (-1) ->> 2 \* (h ((-1)+1)) ->> 2 \* (h 0)  
->> ... ->> 2 \* 4 ->> 8

h (-9) ->> 2 \* (h ((-9)+1))  
->> 2 \* (h (-8)) ->> ... ->> 2048

Die **Auswertung** von **h** terminiert **nicht** sonst:

h 2 ->> 2 \* (h (2+1)) ->> 2 \* (h 3)  
->> 2 \* (2 \* (h (3+1))) ->> 2 \* (2 \* (h 4)) ->> ...

h 3 ->> 2 \* (h (3+1)) ->> 2 \* (h 4) ->> ...

h 9 ->> 2 \* (h (9+1)) ->> 2 \* (h 10) ->> ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

308/170

# Angemessenheit vs. Unangemessenheit für $g$ , $h$

...beide Implementierungen der Funktionen  $g$  und  $h$  sind formal und inhaltlich korrekt, dennoch:

Die Deklarationen:

$g :: \text{Integer} \rightarrow \text{Integer}$	$h :: \text{Integer} \rightarrow \text{Integer}$
$g\ 1 = 2$	$h\ 1 = 2$
$g\ (x+1) = 2 * (g\ x)$	$h\ x = 2 * (h\ (x+1))$

...sind unangemessen, weil intransparent und verschleiern.

$g :: \text{Integer} \rightarrow \text{Integer}$	$h :: \text{Integer} \rightarrow \text{Integer}$
$g\ z$	$h\ z$
$\mid z \geq 1 = 2^z$	$\mid z == 1 = 2$
$\mid \text{otherwise}$	$\mid z \leq 0 = 2^{((\text{abs}\ z)+2)}$
$= \text{error "undefiniert"}$	$\mid \text{otherwise}$
	$= \text{error "undefiniert"}$

...sind angemessen, weil transparent und offen legend.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

309/170

# Kapitel 3.7

## Funktions- und Programmlayout, Abseitsregel

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

**3.7**

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Programmlayout, Programmbedeutung

Für die meisten Programmiersprachen gilt:

- ▶ Die Gestaltung (engl. layout) des Programmtexts beeinflusst
  - ▶ seine Lesbarkeit, Verständlichkeit, Wartbarkeit
  - ▶ aber nicht seine Bedeutung

Nicht so für Haskell – für Haskell gilt:

- ▶ Die Gestalt des Programmtexts trägt Bedeutung!

Dieser Aspekt des Sprachentwurfs

- ▶ ist für Haskell grundsätzlich anders entschieden worden als für Sprachen wie Java, Pascal, C und viele andere.
- ▶ ersetzt `begin/end`- oder `{/}`-Paare durch Gestaltanforderungen.
- ▶ kann als Reminiszenz an Sprachen wie Cobol, Fortran gesehen werden, findet sich aber auch in anderen neueren Sprachen wie z.B. occam.

# Bindungs- und Gültigkeitsbereiche in Haskell

...bestimmt durch **gestaltabhängige Syntax**.

**Eröffnung, Fortsetzung und Beendigung** eines Bindungs- und Gültigkeitsbereichs (Jargon: 'Box') gemäß '**Abseits'-Regel**\*):

- ▶ Das jeweils erste Zeichen einer Deklaration (auch nach **let**, **where**) eröffnet einen neuen Bereich.
- ▶ Ist die nächste Zeile
  - ▶ gegenüber der aktuellen Box nach rechts eingerückt:  
⇒ die aktuelle Zeile wird fortgesetzt
  - ▶ genau am linken Rand der aktuellen Box:  
⇒ eine neue Deklaration innerhalb der aktuellen Box wird eingeleitet
  - ▶ weiter links als die aktuelle Box:  
⇒ die aktuelle Box wird beendet und eine neue eröffnet ('Abseitssituation')

\*) Die **Abseitsregel** findet sich bereits in: Peter J. Landin. *The next 700 Programming Languages*. Communications of the ACM 9(3):157-166, 1966 (S. 160, linke Spalte unten).



# kugel\_OV 'üblich' gestaltet

...Veranschaulichung der [Abseitsregel](#) anhand einer Funktion [kugel\\_OV](#) zur Berechnung von Oberfläche und Volumen einer Kugel mit Radius  $r$ : [Oberfläche](#):  $4\pi r^2$ ; [Volumen](#):  $\frac{4}{3}\pi r^3$ .

```
type Radius      = Float
type Oberflaeche = Float
type Volumen     = Float
pi = 3.14

kugel_OV :: Radius -> (Oberflaeche,Volumen)
kugel_OV r = (oberflaeche r,volumen r)
  where oberflaeche :: Radius -> Oberflaeche
        oberflaeche r = 4 * pi * square r
        volumen :: Radius -> Volumen
        volumen r = (4/3) * pi * cubic r
          where cubic x = x * square x

square :: Float -> Float
square x = x^2
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

313/170

## kugel\_OV korrekt, aber 'unschön' gestaltet

```
type Radius      = Float
type Oberflaeche = Float
type Volumen     = Float

pi = 3.14 :: Float

kugel_OV :: Radius -> (Oberflaeche,Volumen)
kugel_OV r =
  (oberflaeche r,
   volumen r)
  where oberflaeche :: Radius -> Oberflaeche
        oberflaeche r = 4 * pi *
          square r
        volumen :: Radius -> Volumen
        volumen r = (4/3)
          * pi * cubic r
          where cubic x = x * square x

square :: Float -> Float
square x = x^2
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

314/170

# Graphische Veranschaulichung des Box-Begriffs

```
type Radius      = Float -- Radius, Oberflaeche, Volumen und pi
type Oberflaeche = Float -- global im Gesamtprogramm sichtbar
type Volumen     = Float

pi = 3.14 :: Float
```

```
-----
| -- kugelOV global im Gesamtprogramm sichtbar
kugel_OV :: Radius -> (Oberflaeche,Volumen)
kugel_OV =
| (oberflaeche r,
|  volumen r)
|
| -----
| | -- oberflaeche lokal in kugelOV sichtbar
| | where oberflaeche :: Radius -> Oberflaeche
| | oberflaeche r = 4 * pi *
| |   square r
| | ----->
| |
| | -----
| | | -- volumen lokal in kugelOV sichtbar
| | | volumen :: Radius -> Volumen
| | | volumen r = (4/3)
| | |   * pi * cubic r
| | | -----
| | | | -- cubic lokal in volumen sichtbar
| | | | where cubic x = x * square x
| | | | ----->
| | ----->
| ----->
|
| -----
| -- square global im Gesamtprogramm sichtbar
square :: Float -> Float
square x = x^2
| ----->
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

315/170

# Bewährte Layoutkonventionen (1)

...zur Einhaltung der Abseitsregel für Funktionsdefinitionen:

```
funktionsName parameter_1 parameter_2... parameter_n
| waechter_1 = ausdruck_1
| waechter_2 = ausdruck_2
...
| otherwise = ausdruck_k
  where
    v_1 a_1 ... a_n = r_1      -- v_1, v_2,..., sichtbar
    v_2              = r_2      -- in der gesamten Funk-
    ...                       -- tion funktionsName,
                                -- aber nicht außerhalb.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Bewährte Layoutkonventionen (2)

...zum Umgang mit langen Bedingungen und Ausdrücken:

```
funktionsName parameter_1 parameter_2... parameter_n
| waechter_1 = ausdruck_1
| waechter_2 = ausdruck_2
| diesIsteineGanz
  BesondersLangeMehrzeilige
    BedingungAlsWaechter
      = diesIstEinBesonders
        LangerMehrzeiliger
          AusdruckZurWertfestlegung
| waechter_4 = ausdruck_4
...
| otherwise  = ausdruck_k
where...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

317/1700

# Sprachkonstrukt- und Layoutwahl

...nach Angemessenheitserwägungen:

- Was ist gut und einfach lesbar und verständlich?

Illustration:

Vergleiche folgende je drei Implementierungen der Rechen-  
vorschriften miteinander:

- `fib :: Int -> Int`
- `max :: Int -> Int -> Int -> Int`

# Drei Implementierungen für fib

## Mittels Muster:

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

## Mittels bedingter Ausdrücke:

```
fib :: Int -> Int
fib n = if (n == 0) || (n == 1) then 1
        else fib (n-2) + fib (n-1)
```

## Mittels geschachtelter bedingter Ausdrücke:

```
fib :: Int -> Int
fib n = if n == 0
        then 1
        else if n == 1
              then 1
              else fib (n-2) + fib (n-1)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

319/1700

# Drei Implementierungen für max

Mittels geschachtelter bedingter Ausdrücke und anonymer  $\lambda$ -Abstraktion:

```
max :: Int -> Int -> Int -> Int
max = \p q r -> if p>=q then (if p>=r then p else r)
                      else (if q>=r then q else r)
```

Mittels geschachtelter bedingter Ausdrücke:

```
max :: Int -> Int -> Int -> Int
max p q r = if (p>=q) && (p>=r) then p
              else if (q>=p) && (q>=r) then q else r
```

Mittels bewachter Ausdrücke:

```
max :: Int -> Int -> Int -> Int
max p q r
  | (p>=q) && (p>=r) = p
  | (q>=p) && (q>=r) = q
  | otherwise       = r
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

320/1700



Programme können grundsätzlich  
auf zwei Arten geschrieben werden:

So einfach, dass sie offensichtlich keinen Fehler enthalten;  
so kompliziert, dass sie keinen offensichtlichen Fehler enthalten.

C.A.R. 'Tony' Hoare (\* 1934)  
*Turing Award* Preisträger 1980

Gut gewählte Sprachkonstrukte und gut gewähltes Layout

- ▶ unterstützen dabei, Programme einfach und offensichtlich fehlerfrei zu schreiben (vgl. Kap. 3.6)!

In Haskell heißt das

- ▶ 'schönes' Einrücken und zumeist die Verwendung bewachter Ausdrücke und Muster anstelle (geschachtelter) bedingter Ausdrücke (vgl. Kap. 3.1).

# Kapitel 3.8

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

**3.8**

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 3 (1)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 3, Funktionen und Operatoren; Kapitel 4, Rekursion als Entwurfstechnik)
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 2, Expressions, types and values)
-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 1, Elemente funktionaler Programmierung)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3.5, Function types; Kapitel 3.6, Curried functions; Kapitel 4, Defining functions; Kapitel 6, Recursive functions)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7



Kap. 8

Kap. 9

Teil IV

323/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 3 (2)

-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 3, Syntax in Functions; Kapitel 4, Hello Recursion!)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 4, Functional Programming - Partial Function Application and Currying)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 6, Ein bisschen syntaktischer Zucker)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 7, Defining functions over lists)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

324/170

# Kapitel 4

## Typsynonyme, Neue Typen, Typklassen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

**Kap. 4**

4.1

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Kapitel 4.1

## Typsynonyme (type)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

**4.1**

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 4.1.1

## Motivation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

**4.1.1**

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Was bedeuten unsere Daten?

Werte und ihre Typinformation allein erlauben i.a. nur wenig oder keinen Aufschluss darüber, was für Daten sie modellieren bzw. in der äußeren Welt repräsentieren. Betrachte:

```
('A',True)  :: (Char,Bool)
```

```
('Z',False) :: (Char,Bool)
```

```
("Fun",3)   :: (String,Int)
```

```
("Hello",5) :: (String,Int)
```

```
(5.0,8.0,6.8) :: (Float,Float,Float)
```

```
(7.2,9.7,8.7) :: (Float,Float,Float)
```

```
[(5.0,8.0,6.8),(7.2,9.7,8.7),(2.1,4.6,3.6)]  
      :: [(Float,Float,Float)]
```

```
[(2.4,7.9,5.7),(3.2,5.7,4.7),(2.8,9.3,6.7)],  
 (6.3,7.8,7.2)] :: [(Float,Float,Float)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9



# Sprechende Funktionsnamen

...können helfen, die **äußere Semantik** der **modellierten Daten** in Anwendungen **offen zu legen**:

```
erstesZeichenGrossvokal :: String -> (Char,Bool)
erstesZeichenGrossvokal "" = error "Fehler: Leeres Arg."
erstesZeichenGrossvokal (c:_) =
    (c, elem c ['A','E','I','O','U'])
```

```
erstesZeichenGrossvokal "Alpha" ->> ('A',True)
erstesZeichenGrossvokal "Zeta"   ->> ('Z',False)
```

```
echoLaenge :: String -> (String,Int)
echoLaenge s = (s,length s)
```

```
echoLaenge "Fun"    ->> ("Fun",3)
echoLaenge "Hello" ->> ("Hello",5)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

329/170

# Aber nicht immer

...zumindest nicht immer vollständig oder zweifelsfrei:

```
auswertung :: (Float,Float) -> (Float,Float,Float)
auswertung (x,y) = (x,y,geglaettet)
  where geglaettet = (4*x+6*y)/10

auswertung (5.0,8.0) ->> (5.0,8.0,6.8)
auswertung (7.2,9.7) ->> (7.2,9.2,8.7)

reihenausw :: [(Float,Float)] -> [(Float,Float,Float)]
reihenausw [] = []
reihenausw ((x,y) : xys)
    = (auswertung (x,y)) : reihenausw xys

reihenausw [(5.0,8.0),(7.2,9.7),(2.1,4.6)]
    ->> [(5.0,8.0,6.8),(7.2,9.7,8.7),(2.1,4.6,3.6)]
reihenausw [(2.4,7.9),(3.2,5.7),(2.8,9.3)],(6.3,7.8)]
    ->> [(2.4,7.9,5.7),(3.2,5.7,4.7),(2.8,9.3,6.7),
        (6.3,7.8,7.2)]
```

Die äußere Semantik der Gleitkommatupel bleibt verborgen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

330/170

# Was bedeuten also unsere Daten?

Wofür stehen **Gleitkommapaare** und **-tripel** wie

(5.0,8.0)

(7.2,9.7)

(5.0,8.0,**6.8**)

(7.2,9.7,**8.7**)

...und wofür **Listen solcher Paare** und **Tripel**?

[(5.0,8.0), (7.2,9.7), (2.1,4.6)]

[(2.4,7.9,**5.7**), (3.2,5.7,**4.7**), (2.8,9.3,**6.7**),  
(6.3,7.8,**7.2**)]

Für **Aktienkurse**, für **Pegelstände**, für **Positionsdaten**?

# Kapitel 4.1.2

## Typsynonyme

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

**4.1.2**

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Typsynonyme schaffen Abhilfe: Aktienkurse

```
type Kurs          = Float
type Niedrigst     = Kurs
type Hoechst       = Kurs
type Bewertet      = Kurs
type Kursausschlag = (Niedrigst,Hoechst)
type Ausschlagsanalyse = (Niedrigst,Hoechst,Bewertet)
type Kursverlauf   = [Kursausschlag]
type Verlaufsanalyse = [Ausschlagsanalyse]
```

Typsynonymverw. führt jetzt zu sprechenderen Funktionsdef.:

```
auswertung :: Kursausschlag -> Ausschlagsanalyse
auswertung (x,y) = (x,y,geglaettet) -- Moderat risiko-
  where geglaettet = (4*x+6*y)/10    -- freud. Investor

reihenausw :: Kursverlauf -> Verlaufsanalyse
reihenausw [] = []
reihenausw ((x,y) : xys)
  = (auswertung (x,y)) : reihenausw xys
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

333/170

# Typsynonyme schaffen Abhilfe: Aktienkurse

...und Anwendungsaufrufen:

```
ausschlag1 = (5.0,8.0) :: Kursausschlag
```

```
ausschlag2 = (7.2,9.7) :: Kursausschlag
```

```
auswertung ausschlag1 ->> (5.0,8.0,6.8) :: Ausschlagsanalyse
```

```
auswertung ausschlag2 ->> (7.2,9.7,8.7) :: Ausschlagsanalyse
```

```
verlauf1 = [(5.0,8.0),(7.2,9.7),(2.1,4.6)] :: Kursverlauf
```

```
verlauf2 = [(2.4,7.9),(3.2,5.7),(2.8,9.3)] :: Kursverlauf
```

```
reihenausw verlauf1
```

```
->> [(5.0,8.0,6.8),(7.2,9.7,8.7),(2.1,4.6,3.6)]  
      :: Verlaufsanalyse
```

```
reihenausw verlauf2
```

```
->> [(2.4,7.9,5.7),(3.2,5.7,4.7),(2.8,9.3,6.7)]  
      :: Verlaufsanalyse
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Typsynonyme schaffen Abhilfe: Pegelstände

```
type Pegelstand      = Float
type Niedrig         = Pegelstand
type Hoch            = Pegelstand
type Mittel          = Pegelstand
type Messung         = (Niedrig,Hoch)
type Auswertung      = (Niedrig,Hoch,Mittel)
type Messreihe       = [Messung]
type Auswertungsreihe = [Auswertung]
```

Typsynonymverw. ermöglicht jetzt folgende Funktionsdefinitionen:

```
auswertung' :: Messung -> Auswertung
auswertung' (x,y) = (x,y,geglaettet)
  where geglaettet = (x+y)/2      -- Rechn. Mittelwert

reihenausw' :: Messreihe -> Auswertungsreihe
reihenausw' [] = []
reihenausw' ((x,y) : xys)
  = (auswertung' (x,y)) : reihenausw' xys
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

335/170

# Typsynonyme schaffen Abhilfe: Pegelstände

...und Anwendungsaufrufe:

```
mess1 = (5.0,8.0) :: Messung
```

```
mess2 = (7.2,9.7) :: Messung
```

```
auswertung' mess1 ->> (5.0,8.0,6.5) :: Auswertung
```

```
auswertung' mess2 ->> (7.2,9.7,8.45) :: Auswertung
```

```
messreihe1 = [(5.0,8.0),(7.2,9.7),(2.1,4.6)] :: Messreihe
```

```
messreihe2 = [(2.4,7.9),(3.2,5.7),(2.8,9.3)] :: Messreihe
```

```
reihenausw' messreihe1
```

```
->> [(5.0,8.0,6.5),(7.2,9.7,8.45),(2.1,4.6,3.75)]  
:: Auswertungsreihe
```

```
reihenausw' messreihe2
```

```
->> [(2.4,7.9,5.15),(3.2,5.7,4.45),(2.8,9.3,6.05)]  
:: Auswertungsreihe
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9



# Typsynonyme schaffen Abhilfe: Positionsdaten

```
type Koordinate = Float
type X          = Koordinate
type Y          = Koordinate
type Z          = Koordinate
type Ebenenpunkt = (X,Y)
type Raumpunkt   = (X,Y,Z)
type Flaeche     = [Ebenenpunkt]
type Koerper     = [Raumpunkt]
```

Typsynonymverw. ermöglicht folgende Funktionsdefinitionen:

```
auswertung'' :: Ebenenpunkt -> Raumpunkt
auswertung'' (x,y) = (x,y,geglaettet)
  where geglaettet = x+y      -- Summe als Z-Koord.-Wert

reihenausw'' :: Flaeche -> Koerper
reihenausw'' [] = []
reihenausw'' ((x,y) : xys)
  = (auswertung'' (x,y)) : reihenausw'' xys
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

337/170

# Typsynonyme schaffen Abhilfe: Positionsdaten

...und Anwendungsaufrufe:

```
xy1 = (5.0,8.0) :: Ebenenpunkt
```

```
xy2 = (7.2,9.7) :: Ebenenpunkt
```

```
auswertung'' xy1 ->> (5.0,8.0,13.0) :: Raumpunkt
```

```
auswertung'' xy2 ->> (7.2,9.7,16.9) :: Raumpunkt
```

```
flaeche1 = [(5.0,8.0),(7.2,9.7),(2.1,4.6)] :: Flaeche
```

```
flaeche2 = [(2.4,7.9),(3.2,5.7),(2.8,9.3)] :: Flaeche
```

```
reihenausw'' flaeche1
```

```
->> [(5.0,8.0,13.0),(7.2,9.7,16.9),(2.1,4.6,6.7)] :: Koerper
```

```
reihenausw'' flaeche2
```

```
->> [(2.4,7.9,10.3),(3.2,5.7,8.9),(2.8,9.3,12.1)] :: Koerper
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

338/170

# Kapitel 4.1.3

## Tupeltypsynonyme und Selektorfunktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

**4.1.3**

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Selektorfunktionen für Tupeltypen

...oder kurz **Selektoren** für **Tupeltypen** wie **Kursausschlag**, **Ausschlagsanalyse**, **Messung**, **Auswertung**, **Ebenenpunkt**, **Raumpunkt** sind in Haskell

- ▶ vordefiniert für **Paartypen** (**fst**, **snd**) (s. **Kap. 2.2.1**)
- ▶ selbst zu definieren für **höherstellige Tupeltypen**.

Generell gilt: **Musterbasierte Definitionen** sind für **Selektoren** meist am zweckmäßigsten.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

340/170

# Selektoren für Wertpapierdaten

...in **musterbasierter** Definitionsweise:

```
ndgstKurs :: Kursausschlag -> Niedrigst
```

```
ndgstKurs (ndgst,_) = ndgst
```

```
hchstKurs :: Kursausschlag -> Hoechst
```

```
hchstKurs (_,hchst) = hchst
```

```
ndgstKursA :: Ausschlagsanalyse -> Niedrigst
```

```
ndgstKursA (ndgst,_,_) = ndgst
```

```
hchstKursA :: Ausschlagsanalyse -> Hoechst
```

```
hchstKursA (_,hchst,_) = hchst
```

```
bwKursA :: Ausschlagsanalyse -> Bewertet
```

```
bwKursA (_,_,bwk) = bwk
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

341/170

# In gleicher Weise

...lassen sich Selektoren für

- ▶ Pegelstandsdaten
- ▶ Positionsdaten

definieren.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

**4.1.3**

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Selektoren für Pegelstandsdaten

...in **musterbasierter** Definitionsweise:

```
nPglM :: Messung -> Niedrig
```

```
nPglM (n,_) = n
```

```
hPglM :: Messung -> Hoch
```

```
hPglM (_,h) = h
```

```
nPglA :: Auswertung -> Niedrig
```

```
nPglA (n,_,_) = n
```

```
hPglA :: Auswertung -> Hoch
```

```
hPglA (_,h,_) = h
```

```
mPglA :: Auswertung -> Mittel
```

```
mPglA (_,_,m) = m
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Selektoren für Positionsdaten

...in **musterbasierter** Definitionsweise:

$\text{xE} :: \text{Ebenenpunkt} \rightarrow X$

$\text{xE} (x, \_) = x$

$\text{yE} :: \text{Ebenenpunkt} \rightarrow Y$

$\text{yE} (\_, y) = y$

$\text{xR} :: \text{Raumpunkt} \rightarrow X$

$\text{xR} (x, \_, \_) = x$

$\text{yR} :: \text{Raumpunkt} \rightarrow Y$

$\text{yR} (\_, y, \_) = y$

$\text{zR} :: \text{Raumpunkt} \rightarrow Z$

$\text{zR} (\_, \_, z) = z$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9



# Die Selektoren

...für **Kursverläufe**, **Messungen** und **Ebenenpunkte** alternativ (punktfrei) auf die vordefinierten Parselektoren abgestützt:

```
ndgstKurs :: Kursausschlag -> Niedrigst
```

```
ndgstKurs = fst
```

```
hchstKurs :: Kursausschlag -> Hoechst
```

```
hchstKurs = snd
```

```
nPg1M :: Messung -> Niedrig
```

```
nPg1M = fst
```

```
hPg1M :: Messung -> Hoch
```

```
hPg1M = snd
```

```
xE :: Ebenenpunkt -> X
```

```
xE = fst
```

```
yE :: Ebenenpunkt -> Y
```

```
yE = snd
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

345/170

# Kapitel 4.1.4

## Weitere Beispiele

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

**4.1.4**

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Zwei weitere Beispiele

...für **Tupeltypsynonyme** und zugehörige **Selektoren**:

- ▶ **Studentendaten**
- ▶ **Buchhandelsdaten**

mit Präfixverwendung der Tupelkonstruktoren **(,,)** und **(,,,)** zur Kreierung von Vier- und Fünftupeln.

# Typsynonyme u. Selektoren f. Studentendaten

```
type Vorname      = String
type Nachname     = String
type Email        = String
type Studienkennzahl = Int
type Skz          = Studienkennzahl
type Student      = (Vorname, Nachname, Email, Skz)
```

```
((,,) "Max Muster" "e123456@stud.tuwien.ac.at" 534
->> ("Max", "Muster", "e123456@stud.tuwien.ac.at", 534)
                                     :: Student
```

```
vorname :: Student -> Vorname           (Ausschließlich
vorname (v,n,e,k) = v                   Variablenmuster)
```

```
nachname :: Student -> Nachname
nachname (v,n,e,k) = n
```

```
email :: Student -> Email
email (v,n,e,k) = e
```

```
skz :: Student -> Studienkennzahl
skz (v,n,e,k) = k
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Typsynonyme u. Selektoren f. Buchhandelsdat.

```
type Autor    = String
type Titel    = String
type Auflage  = Int
type Jahr     = Int
type Lagernd  = Bool
type Buch     = (Autor,Titel,Auflage,Jahr,Lagernd)

(,,,,"Simon Thompson" "Haskell" 3 2011 True
->> ("Simon Thompson","Haskell",3,2011,True) :: Buch

autor :: Buch -> Autor                (Variablenmuster
                                     und 'wild card')
autor (a,_,_,_,_) = a

titel :: Buch -> Titel
titel (_,t,_,_,_) = t

auflage :: Buch -> Auflage
auflage (_,_,a,_,_) = a

erschienen :: Buch -> Jahr
erschienen (_,_,_,j,_) = j

lagernd :: Buch -> Lagernd
lagernd (_,_,_,_,l) = l
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

349/170

# Kapitel 4.1.5

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

**4.1.5**

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Zusammenfassung

## Typsynonyme

- ▶ erlauben die **äußere Semantik von Datentypen** durch Wahl eines **guten und sprechenden Namens** offenzulegen und mitzuteilen.
- ▶ ermöglichen damit **aussagekräftigere** und **sprechendere** Funktionssignaturen.
- ▶ erhöhen dadurch die **Lesbarkeit, Verständlichkeit** und **Transparenz** von und in Programmen.

## Aber: Typsynonyme

- ▶ führen **keine neuen Typen** ein, sondern ausschließlich **neue Namen** für bereits existierende Typen, sog. **Alias-Namen** oder **Synonyme**.
- ▶ leisten daher **keinen Beitrag zu mehr Typsicherheit**.

Dazu mehr in **Kapitel 4.2** und **Kapitel 5**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.1.1

4.1.2

4.1.3

4.1.4

4.1.5

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

351/170

# Kapitel 4.2

## Neue Typen (newtype)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

**4.2**

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

352/170



# Kapitel 4.2.1

## Motivation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

**4.2.1**

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

353/170

# Neue Typen

## Typsynonyme

- ▶ führen **keine neuen Typen** ein, sondern **neue Namen** für bereits existierende Typen, sog. **Aliasnamen** oder **Synonyme**.
- ▶ leisten daher **keinen Beitrag zu mehr Typsicherheit**.

Warum?

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

354/170

# Neue Typen

## Typsynonyme

- ▶ dürfen überall dort stehen und verwendet werden, wo auch ihre jeweiligen Grundtypen stehen dürfen und umgekehrt.

In anderen Worten:

## Typsynonyme und ihre jeweiligen Grundtypen

- ▶ dürfen sich ohne Einschränkung **wechselweise vertreten**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

355/170

# Datenzusammengehörigkeit

...im Sinn **intendierter Typbedeutung** ist **vertikal** ausgedrückt:

Grundtyp	Durch Typsynonym intendierter Typ		
	Wertpapierdaten	Pegeldaten	Positionsdaten
Float	Kurs Niedrigst Hoechst Bewertet	Pegelstand Niedrig Hoch Mittel	Koordinate X Y Z
(Float,Float)	Kursausschlag (ndgst,hchst)	Messung (ndg,hch)	Ebenenpunkt (x,y)
(Float,Float,Float)	Ausschlagsanalyse (ndgst,hchst,gg)	Auswertung (ndg,hch,mtt)	Raumpunkt (x,y,z)
[(Float,Float)]	Kursverlauf Kursausschlag-L.	Messreihe Mess'gen-L.	Fläche E'punkt-Liste
[(Float,Float,Float)]	Verlaufsanalyse Ausschlagsa'yse-L.	Ausw'gsreihe Ausw'gs-L.	Körper R'punkt-Liste

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Datenzusammengehörigkeit

...im Sinn tatsächlicher Typbedeutung ist horizontal ausgedrückt:

Die (jeweils gleichfarbigen) Typ (-bezeichner)

- ▶ Float mit seinen Aliassen  
Kurs, Niedrigst, Hoechst, Bewertet,  
Pegelstand, Niedrig, Hoch, Mittel,  
Koordinate, X, Y, Z
- ▶ (Float,Float) mit seinen Aliassen  
Kursausschlag, Messung, Ebenenpunkt
- ▶ (Float,Float,Float) mit seinen Aliassen  
Ausschlagsanalyse, Auswertung, Raumpunkt
- ▶ [(Float,Float)] mit seinen Aliassen  
Kursverlauf, Messreihe, Flaeche
- ▶ [(Float,Float,Float)] mit seinen Aliassen  
Verlaufsanalyse, Auswertungsreihe, Koerper

...dürfen einander (im Widerspruch zu ihrer intendierten Bedeutung) wechselweise vertreten.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

357/170

# Typisierung aus Haskell-Sicht (1)

...für die **Typisierung** gilt deshalb (trotz/wegen der Verwendung von **Typsynonymen**):

```
Kurs, Niedrigst, Hoechst, Bewertet,  
Pegelstand, Hoch, Niedrig, Mittel,  
Koordinate, X, Y, Z :: Float
```

```
Kursausschlag, Messung, Ebenenpunkt :: (Float,Float)
```

```
Ausschlagsanalyse, Auswertung, Raumpunkt  
:: (Float,Float,Float)
```

```
Kursverlauf, Messreihe, Flaeche :: [(Float,Float)]
```

```
Verlaufsanalyse, Auswertungsreihe, Koerper  
:: [(Float,Float,Float)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

358/170

# Typisierung aus Haskell-Sicht (2)

...sowie für die darauf aufbauenden **Verarbeitungsfunktionen**:

Wertpapierdaten:

```
auswertung :: (Float,Float) -> (Float,Float,Float)
reihenausw :: [(Float,Float)] -> [(Float,Float,Float)]
```

Wasserstandsdaten:

```
auswertung' :: (Float,Float) -> (Float,Float,Float)
reihenausw' :: [(Float,Float)] -> [(Float,Float,Float)]
```

Positionsdaten:

```
auswertung'' :: (Float,Float) -> (Float,Float,Float)
reihenausw'' :: [(Float,Float)] -> [(Float,Float,Float)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

359/170

# Konsequenz (1)

...aufgrund der jeweiligen wechselseitigen Typgleichheiten von

- ▶ `auswertung`, `auswertung'`, `auswertung''`
- ▶ `reihenausw`, `reihenausw'`, `reihenausw''`

und der jeweiligen wechselseitigen Typgleichheiten von

- ▶ `Kursausschlag`, `Messung`, `Ebenenpunkt`
- ▶ `Kursverlauf`, `Messreihe`, `Flaeche`



## Konsequenz (2)

...arbeiten die Funktionen

- ▶ `auswertung`, `auswertung'`, `auswertung''`

typfehlerfrei auf jedem Wert der Typen

- ▶ `Kursausschlag`, `Messung`, `Ebenenpunkt`

Ebenso typfehlerfrei arbeiten die Funktionen

- ▶ `reihenausw`, `reihenausw'`, `reihenausw''`

auf jedem Wert der Typen

- ▶ `Kursverlauf`, `Messreihe`, `Flaeche`

Daten können so Argument von Funktionen werden, für die das gemäß der mit den Typsynonymen ausgedrückten intendierten Semantik nicht möglich sein sollte.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

361/170

# Typsicherheit sieht anders aus (1)

Die Funktionen zur Wertpapierdatenverarbeitung sind typfehlerfrei auch auf Wasserstands- und Positionsdaten anwendbar:

## Wasserstandsdaten

```
auswertung mess1 ->> (5.0,8.0,6.8)
auswertung mess2 ->> (7.2,9.7,8.7)

reihenausw messreihe1
->> [(5.0,8.0,6.8),(7.2,9.7,8.7),(2.1,4.6,3.6)]
reihenausw messreihe2
->> [(2.4,7.9,5.7),(3.2,5.7,4.7),(2.8,9.3,6.7)]
```

## Positionsdaten

```
auswertung xy1 ->> (5.0,8.0,6.8)
auswertung xy2 ->> (7.2,9.7,8.7)

reihenausw flaeche1
->> [(5.0,8.0,6.8),(7.2,9.7,8.7),(2.1,4.6,3.6)]
reihenausw flaeche2
->> [(2.4,7.9,5.7),(3.2,5.7,4.7),(2.8,9.3,6.7)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

362/170

# Typsicherheit sieht anders aus (2)

Die Funktionen zur **Wasserstandsdatenverarbeitung** sind typfehlerfrei auch auf **Wertpapier**- und **Positionsdaten** anwendbar:

## Wertpapierdaten

```
auswertung' ausschlag1 ->> (5.0,8.0,6.5)
auswertung' ausschlag2 ->> (7.2,9.7,8.45)

reihenausw' verlauf1
->> [(5.0,8.0,6.5),(7.2,9.7,8.45),(2.1,4.6,3.35)]
reihenausw' verlauf2
->> [(2.4,7.9,5.15),(3.2,5.7,4.45),(2.8,9.3,6.05)]
```

## Positionsdaten

```
auswertung' xy1 ->> (5.0,8.0,6.5)
auswertung' xy2 ->> (7.2,9.7,8.45)

reihenausw' flaeche1
->> [(5.0,8.0,6.5),(7.2,9.7,8.45),(2.1,4.6,3.35)]
reihenausw' flaeche2
->> [(2.4,7.9,5.15),(3.2,5.7,4.45),(2.8,9.3,6.05)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

363/170

# Typsicherheit sieht anders aus (3)

Die Funktionen zur **Positionsdatenverarbeitung** sind typfehlerfrei auch auf **Wertpapier-** und **Wasserstandsdaten** anwendbar:

## Wertpapierdaten

```
auswertung'' ausschlag1 ->> (5.0,8.0,13.0)
auswertung'' ausschlag2 ->> (7.2,9.7,16.9)

reihenausw'' verlauf1
->> [(5.0,8.0,13.0),(7.2,9.7,16.9),(2.1,4.6,6.7)]
reihenausw'' verlauf2
->> [(2.4,7.9,10.3),(3.2,5.7,8.9),(2.8,9.3,12.1)]
```

## Wasserstandsdaten

```
auswertung'' mess1 ->> (5.0,8.0,13.0)
auswertung'' mess2 ->> (7.2,9.7,16.9)

reihenausw'' messreihe1
->> [(5.0,8.0,13.0),(7.2,9.7,16.9),(2.1,4.6,6.7)]
reihenausw'' messreihe2
->> [(2.4,7.9,10.3),(3.2,5.7,8.9),(2.8,9.3,12.1)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

364/170

# Weit reichende Folgen fehlender Typsicherheit

Was mit der Verarbeitung von

- ▶ Wertpapierdaten, Wasserständen, Positionsdaten

möglich ist, ist auch mit Werten anderer **Typsynonyme** möglich, z.B. für:

- ▶ Meilen/Kilometer, Pferdestärken/Kilowatt, Kraftpfund (engl. pound-force, lb, lbf))/Newton, etc.

```
type Meile      = Float
type Kilometer  = Float
type PS         = Float
...
```

Eine **Petitesse**? Fragen Sie die **NASA**!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

365/170

# Marssonde 'Climate Orbiter'

## Ereignis

- ▶ Totalverlust der Sonde **Mars Climate Orbiter** am 23.09.1998 beim Versuch in die Marsatmosphäre einzutauchen.

## Ursache

- ▶ Widersprüchliche Verwendung metrischer und nichtmetrischer Daten zwischen **Lockheed Martin Astronautics** und **NASA-Labor Jet Propulsion Laboratory (JPL)**:  
**Pound-force** vs. **Newton** und weiterer **metrischer** und **SI-Einheiten**.

## Schadenssumme

- ▶ Rd. 125 Millionen US\$.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

366/170

# Marssondendebakel, Hintergrundmaterial

## NASA-Datenbank 'Lessons Learned':

[http://www-bcf.usc.edu/~meshkati/fea03/appendix.html/  
Mars%20Climate%20Orbiter%20NASA%20LLIS%Database.htm](http://www-bcf.usc.edu/~meshkati/fea03/appendix.html/Mars%20Climate%20Orbiter%20NASA%20LLIS%Database.htm)

## NASA-Berichte zu Sondenverlust (...0930) und Ursache (...1110):

<http://mars.nasa.gov/msp98/news/mco990930.html>

<http://mars.nasa.gov/msp98/news/mco991110.html>

## Weitere zusammenfassende Berichte (...0052) und über ignorierte Warnhinweise in der Anflugphase (...report/):

[http://www.sciencedirect.com/science/article/  
pii/S0263786309000052](http://www.sciencedirect.com/science/article/pii/S0263786309000052)

[https://www.wired.com/2010/11/  
1110mars-climate-observer-report/](https://www.wired.com/2010/11/1110mars-climate-observer-report/)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

367/170

# Typsynonyme (1)

...verhindern nicht, die sprichwörtlichen Äpfel und Birnen

```
type Apfel = String
type Birne = String

jonathan = "Jonathan" :: Apfel
williams = "Williams" :: Birne
apfel     = "Williams" :: Apfel
birne     = "Jonathan" :: Birne
```

...‘erfolgreich’ miteinander zu vergleichen:

```
[apfel,jonathan] == [williams,birne] ->> True
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

368/170



# Typsynonyme (2)

...erlauben **intendierte Typ- und Wertbenutzungen anzuzeigen**,

- ▶ **nicht aber durchzusetzen.**

**Typsynonyme** können deshalb **nicht verhindern**, Funktionen

- ▶ **dezidiert** zur Verarbeitung von z.B. Wertpapier-, Wasserstands- und Positionsdaten

versehentlich, irrtümlich oder auch absichtlich auf

- ▶ **ungeeignete und nicht dafür vorgesehene Daten** anzuwenden.

# Übertragen auf unser durchgehendes Beispiel

...zwischen **Aktienkursen**, **Pegelständen** und **Positionsangaben**

- ▶ besteht **kein innerer** oder sonstiger **bedeutungsmäßiger Zusammenhang**.
- ▶ Es gibt **keinen vernünftigen Grund**, Funktionen für Wertpapierdaten auf Wasserstandsdaten oder Positionsdaten anzuwenden und umgekehrt.
- ▶ Eine **gute Programmiersprache** sollte es ermöglichen, ungeeignete Verwendung auszuschließen:
  - ▶ Das programmiersprachliche Mittel dazu: **Typsysteme!**
  - ▶ Im Typsystem von **Haskell**: **Neue Typen** (und als Verallgemeinerung **algebraische Datentypen**, siehe **Kap. 5**).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

370/170

# Kapitel 4.2.2

## Neue Typen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

**4.2.2**

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

371/170

# Neue Typen, newtype-Deklarationen

Wir ersetzen **Typsynonym-Deklarationen**:

```
type Kurs          = Float
type Pegelstand    = Float
type Koordinate     = Float
```

durch **newtype**-Deklarationen:

```
newtype Kurs          = K Float
newtype Pegelstand    = Pgl Float
newtype Koordinate     = Koordinate Float
```

*Diagrammatische Anmerkungen:*

- Über `Kurs`: *Typname*
- Über `K`: *Nutzdatentyp*
- Über `Pgl`: *Datenkonstruktorname*

**Beachte:** *Typname* und *Datenkonstruktorname* dürfen übereinstimmen; sie sind durch den Anwendungskontext unterscheidbar.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

372/170

# Alles andere bleibt (syntaktisch) gleich

...am Beispiel für Wertpapierdaten:

```
newtype Kurs          = K Float
type Niedrigst         = Kurs
type Hoechst          = Kurs
type Bewertet          = Kurs
type Kursausschlag     = (Niedrigst,Hoechst)
type Ausschlagsanalyse = (Niedrigst,Hoechst,Bewertet)
type Kursverlauf       = [Kursausschlag]
type Verlaufsanalyse   = [Ausschlagsanalyse]
```

Beachte allerdings:

- ▶ Wie `Kurs` sind auch `Niedrigst`, `Hoechst` und `Bewertet` keine Synonyme mehr für `Float`, sondern für den neuen `Typ Kurs`.
- ▶ In gleicher Weise sind auch `Kursausschlag`, `Ausschlagsanalyse`, `Kursverlauf` und `Verlaufsanalyse` keine Synonyme mehr für `Float`-Paare, `Float`-Tripel und Listen von `Float`-Paaren und `Float`-Tripeln.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

373/170

# Anpassung der Funktionsdefinitionen

..**Konstrukoren** in Argumentmustern, alles andere (syntaktisch) gleich:

```
auswertung :: Kursausschlag -> Ausschlagsanalyse
auswertung (K x,K y) = (K x,K y,K geglaettet)
  where geglaettet = (4*x+6*y)/10

reihenausw :: Kursverlauf -> Verlaufsanalyse
reihenausw [] = []
reihenausw ((K x,K y)) : xys
    = (auswertung (K x, K y)) : reihenausw xys
```

statt (wie mit Typsynonymen in Kapitel 4.1):

```
auswertung :: Kursausschlag -> Ausschlagsanalyse
auswertung (x,y) = (x,y,geglaettet)
  where geglaettet = (4*x+6*y) / 10

reihenausw :: Kursverlauf -> Verlaufsanalyse
reihenausw [] = []
reihenausw ((x,y) : xys)
    = (auswertung (x,y)) : reihenausw xys
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

374/170

# Analog für Wasserstandsdaten

Modifizierte Typdeklarationen:

```
newtype Pegelstand    = Pgl Float
type Niedrig          = Pegelstand
type Hoch             = Pegelstand
type Mittel           = Pegelstand
type Messung          = (Niedrig,Hoch)
type Auswertung       = (Niedrig,Hoch,Mittel)
type Messreihe        = [Messung]
type Auswertungsreihe = [Auswertung]
```

Angepasste Funktionsdefinitionen:

```
auswertung' :: Messung -> Auswertung
auswertung' (Pgl x,Pgl y) = (Pgl x,Pgl y,Pgl geglaettet)
  where geglaettet = (x+y)/2

reihenausw' :: Messreihe -> Auswertungsreihe
reihenausw' [] = []
reihenausw' ((Pgl x,Pgl y) : xys)
  = (auswertung' (Pgl x,Pgl y)) : reihenausw' xys
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

# Analog für Positionsdaten

Modifizierte Typdeklarationen:

```
newtype Koordinate = Koordinate Float
type X              = Koordinate
type Y              = Koordinate
type Z              = Koordinate
type Ebenenpunkt   = (X,Y)
type Raumpunkt      = (X,Y,Z)
type Flaechen       = [Ebenenpunkt]
type Koerper        = [Raumpunkt]
```

Angepasste Funktionsdefinitionen:

```
auswertung'' :: Ebenenpunkt -> Raumpunkt
auswertung'' (Koordinate x,Koordinate y)
  = (Koordinate x,Koordinate y,Koordinate geglaettet)
  where geglaettet = x+y
```

```
reihenausw'' :: Flaechen -> Koerper
reihenausw'' [] = []
reihenausw'' ((Koordinate x,Koordinate y) : xys)
  = (auswertung'' (Koordinate x,Koordinate y)) : reihenausw'' xys
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

376/1700



# Kapitel 4.2.3

## Typsicherheit erreicht

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

**4.2.3**

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

377/170

# Typsicherheit erreicht!

...die (Nutz-) Daten, die Gleitkommawerte, liegen jetzt geschützt hinter Datenkonstruktoren:

## Wertpapierdaten:

```
ausschlag = (K 5.0, K 8.0)
verlauf   = [(K 5.0, K 8.0), (K 7.2, K 9.7), (K 2.1, K 4.6)]
```

## Wasserstandsdaten:

```
messung    = (Pgl 5.0, Pgl 8.0)
messreihe  = [(Pgl 5.0, Pgl 8.0), (Pgl 7.2, Pgl 9.7),
              (Pgl 2.1, Pgl 4.6)]
```

## Positionsdaten:

```
ebenenpunkt = (Koordinate 5.0, Koordinate 8.0)
flaeche      = [(Koordinate 5.0, Koordinate 8.0),
                (Koordinate 7.2, Koordinate 9.7),
                (Koordinate 2.1, Koordinate 4.6)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

378/170

# Wertpapierdaten typgesichert typsicher!

Wertpapierdatenverarbeitungsfunktionen:

```
auswertung :: Kursausschlag -> Ausschlagsanalyse  
reihenausw :: Kursverlauf -> Verlaufsanalyse
```

Anwendbarkeit auf Wertpapierdaten wie gewünscht:

```
auswertung ausschlag ->> (K 5.0,K 8.0,K 6.8)  
reihenausw verlauf    ->> [(K 5.0,K 8.0,K 6.8),  
                           (K 7.2,K 9.7,K 8.7),  
                           (K 2.1,K 4.6,K 3.6)]
```

Keine Anwendbarkeit auf Wasserstands-, Positionsdaten oder Gleitkommazahlen:

```
auswertung messung      ->> "Fehler: Typen passen nicht."  
reihenausw messreihe    ->> "Fehler: Typen passen nicht."  
  
auswertung ebenenpunkt ->> "Fehler: Typen passen nicht."  
reihenausw flaeche      ->> "Fehler: Typen passen nicht."  
  
auswertung (5.0,8.0)    ->> "Fehler: Typen passen nicht."  
reihenausw [5.0,8.0),(7.2,9.7),(2.1,4.6)]  
                                ->> "Fehler: Typen passen nicht."
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.2.1

4.2.2

4.2.3

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

379/1700

# In gleicher Weise jetzt auch

...Wasserstands- und Positionsdaten jetzt typgesichert typsicher!

Wasserstandsdaten:

```
auswertung' :: Messung    -> Auswertung
reihenausw'  :: Messreihe -> Auswertungsreihe
```

Positionsdaten:

```
auswertung'' :: Ebenenpunkt -> Raumpunkt
reihenausw'' :: Flaeche     -> Koerper
```

# Mission erfüllt: Typsicherheit erreicht!

Das Konzept **neuer Typen**, **Nutzdaten** hinter **Datenkonstruktoren** zu verbergen und **schützen**, erlaubt

- ▶ zusätzlich zum Anzeigen **intendierter Typ- und Wertbenutzungen**

diese auch **durchzusetzen** und **(Nutz-) Daten** so vor

- ▶ **versehentlicher** wie **absichtlicher** oder **böswilliger Verarbeitung** durch nicht dafür vorgesehene Funktionen

**wirksam** zu **schützen**.

# Kapitel 4.3

## Typklassen (class)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

**4.3**

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 4.3.1

## Motivation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

**4.3.1**

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Typsicherheit erreicht. Mission erfüllt? (1)

Betrachte folgende Wertpapierdaten:

- Typgesichert (`newtype Kurs = K Float`):

```
gs_kurs1      = K 5.0
gs_kurs2      = K 7.2
gs_ausschlag1 = (K 5.0, K 8.0)
gs_ausschlag2 = (K 7.2, K 9.7)
gs_verlauf1   = [(K 5.0, K 8.0), (K 7.2, K 9.7),
                  (K 2.1, K 4.6)]
gs_verlauf2   = [(K 2.4, K 7.9), (K 3.2, K 5.7),
                  (K 2.8, K 9.3)]
```

- Typungesichert (`type Kurs = Float`):

```
ugs_kurs1      = 5.0
ugs_kurs2      = 7.2
ugs_ausschlag1 = (5.0, 8.0)
ugs_ausschlag2 = (7.2, 9.7)
ugs_verlauf1   = [(5.0, 8.0), (7.2, 9.7), (2.1, 4.6)]
ugs_verlauf2   = [(2.4, 7.9), (3.2, 5.7), (2.8, 9.3)]
```



# Typsicherheit erreicht. Mission erfüllt? (2)

...und die Ergebnisse folgender Ausdrucksvergleiche:

```
ugs_kurs1 == ugs_kurs1 ->> True
```

```
ugs_kurs1 /= ugs_kurs2 ->> True
```

```
gs_kurs1 == gs_kurs1 ->> "Fehler: (==) unbekannt"
```

```
gs_kurs1 /= gs_kurs2 ->> "Fehler: (/=) unbekannt"
```

```
ugs_ausschlag1 == ugs_ausschlag1 ->> True
```

```
ugs_ausschlag1 /= ugs_ausschlag2 ->> True
```

```
gs_ausschlag1 == gs_ausschlag1 ->> "Fehler: (==) unbekannt"
```

```
gs_ausschlag1 /= gs_ausschlag2 ->> "Fehler: (/=) unbekannt"
```

```
ugs_verlauf1 == ugs_verlauf1 ->> True
```

```
ugs_verlauf1 /= ugs_verlauf2 ->> True
```

```
gs_verlauf1 == gs_verlauf1 ->> "Fehler: (==) unbekannt"
```

```
gs_verlauf1 /= gs_verlauf2 ->> "Fehler: (/=) unbekannt"
```

Schutz vor missbräuchlicher Nutzung **überschießend**? Auch definitiv gewollte Zugriffe und Verarbeitung jetzt nicht mehr möglich?

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Ad hoc Abhilfe (1)

Definiere Gleichheitstests für Kurse, Kursausschläge, Kursverläufe:

```
k_eq :: Kurs -> Kurs -> Bool  
(K k1) 'k_eq' (K k2) = k1==k2
```

```
k_neq :: Kurs -> Kurs -> Bool  
k1 'k_neq' k2 = not (k1 'k_eq' k2)
```

```
ka_eq :: Kursausschlag -> Kursausschlag -> Bool  
(K k1,K k2) 'ka_eq' (K k3,K k4) = (k1,k2)==(k3,k4)
```

```
ka_neq :: Kursausschlag -> Kursausschlag -> Bool  
ka1 'ka_neq' ka2 = not (ka1 'ka_eq' ka2)
```

```
kv_eq :: Kursverlauf -> Kursverlauf -> Bool  
[] 'kv_eq' [] = True  
(ka:kas) 'kve_q' (la:las) = ka 'ka_eq' la && kas 'kv_eq' las  
_ 'kv_eq' _ = False
```

```
kv_neq :: Kursverlauf -> Kursverlauf -> Bool  
kv1 'kv_neq' kv2 = not (kv1 'kv_eq' kv2)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Ad hoc Abhilfe (2)

Damit ergeben sich folgende Vergleichsergebnisse:

```
gs_kurs1      'k_eq'   gs_kurs1      ->> True
gs_kurs1      'k_neq'  gs_kurs2      ->> True
gs_ausschlag1 'ka_eq'   gs_ausschlag1 ->> True
gs_ausschlag1 'ka_neq' gs_ausschlag2 ->> True
gs_verlauf1   'kv_eq'   gs_verlauf1   ->> True
gs_verlauf1   'kv_neq'  gs_verlauf2   ->> True
```

⇒ *Ad hoc* Abhilfe erfüllt den Zweck.

# Ad hoc Abhilfe generalisierbar?

## Gleichheits- und Ungleichheitstests

- ▶ könnten jetzt in gleicher Weise für **Wasserstands-** und **Positionsdaten** definiert werden.

Jedoch wäre dies:

- ▶ **Aufwändig, unpraktisch, wenig elegant** (allein die Wahl der vielen Namen wäre bereits in hohem Maß unschön und mühsam).

**Haskell** bietet ein **zweckmäßigeres und schlagkräftigeres Sprachmittel** an:

- ▶ **Typklassen** (**vordefiniert** und **selbstdefiniert**)

# Kapitel 4.3.2

## Vordefinierte Typklassen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

**4.3.2**

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Typklassen in Haskell

Typklassen (gleich ob **vor-** oder **selbstdefiniert**)

- ▶ haben **Typen als Elemente**.
- ▶ legen eine Menge von **Operationen und Relationen** fest, die auf den Werten ihrer Elemente implementiert sein müssen.
- ▶ können für diese Operationen und Relationen bereits vollständige **Standardimplementierungen** oder noch zu vervollständigende **Protoimplementierungen** vorsehen.

**Typen**

- ▶ werden durch **Instanzbildung** zu Elementen bzw. Instanzen einer Typklasse.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Beispiel 1: Vordefinierte Typklasse Eq

...für Typen, deren Werte absolut verglichen werden können, d.h. auf Gleichheit, Ungleichheit:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x==y)    -- Protoimplementierungen
  x == y = not (x/=y)    -- für (/=) und (==)
```

## Die Typklasse Eq

- ▶ verlangt von Instanzen die Implementierung von zwei Wahrheitswertfunktionen (oder Prädikaten): `(==)`, `(/=)`.
- ▶ stellt für beide Wahrheitswertfunktion eine Protoimplementierung zur Verfügung.

## Minimalvervollständigung bei Instanzbildungen für Eq:

- ▶ Implementierung von entweder `(==)` oder `(/=)`.

# Bemerkung

Die **Protoimplementierungen** für sich allein sind

- ▶ unvollständig und nicht ausreichend, da sie sich wechselseitig aufeinander abstützen.

Dennoch ergibt sich folgender **Vorteil** aus ihrer Angabe:

- ▶ Bei Instanzbildungen reicht es, entweder eine Implementierung für **(==)** oder für **(/=)** anzugeben und deren Protoimplementierung dadurch zu **überschreiben**. Für den jeweils anderen Operator ist seine Protoimplementierung dadurch automatisch vollständig.
- ▶ Auch für beide Funktionen können bei der Instanzbildung Implementierungen angegeben werden. In diesem Fall werden beide Protoimplementierungen **überschrieben**.



# Instanzbildungen für die Typklasse Eq (1)

...am Beispiel des Typs `Bool` der Wahrheitswerte:

```
instance Eq Bool where
  True  == True  = True      -- Überschreiben der
  False == False = True      -- Proto-Impl. von (==)
  _     == _     = False
```

Alternativ, `gleichwertig`:

```
instance Eq Bool where
  True  /= True  = False     -- Überschreiben der
  False /= False = False     -- Proto-Impl. von (/=)
  _     /= _     = True
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Instanzbildungen für die Typklasse Eq (2)

...am Beispiel eines Typs `Punkt` für Punkte in der reellwertigen  $(x,y)$ -Ebene:

```
type X_Koord  = Float
type Y_Koord  = Float
newtype Punkt = Pkt (X_Koord,Y_Koord)

instance Eq Punkt where
    (Pkt (u,v)) == (Pkt (u',v')) = (u==u') && (v==v')
```

...Punkte sind 'gleich', wenn sie in x- **und** y-Koordinate übereinstimmen, d.h. **gleich** sind.

Beachte:

- ▶ Gleichheit von `Punkt`-Werten wird zurückgeführt auf Gleichheit von `Float`-Werten.
- ▶ Sprechweise: Das Relatorsymbol `(==)` wird (durch die Instanzbildung) **überladen**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Instanzbildungen für die Typklasse Eq (3)

Auch andere Auffassungen von 'Gleichheit' von Punkten wären möglich, z.B. folgende schwächere Gleichheitsauffassung:

```
instance Eq Punkt where  
  (Pkt (u,v)) == (Pkt (u',v')) = (u==u') || (v==v')
```

...Punkte sind 'gleich', wenn sie in x- **oder** y-Koordinate übereinstimmen, d.h. auf derselben Parallele zur y- oder x-Achse liegen.

# Beispiel 2: Vordefinierte Typklasse Ord

...für Typen, deren Werte relativ verglichen werden können:

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min :: a -> a -> a
  compare x y -- Protoimplementierungen
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT
  x <= y        = compare x y /= GT
  x < y         = compare x y == LT
  x >= y        = compare x y /= LT
  x > y         = compare x y == GT
  max x y      = Vergleiche auf Ordering-Werten
    | x <= y    = y
    | otherwise = x
  min x y
    | x <= y    = x
    | otherwise = y
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Bemerkung

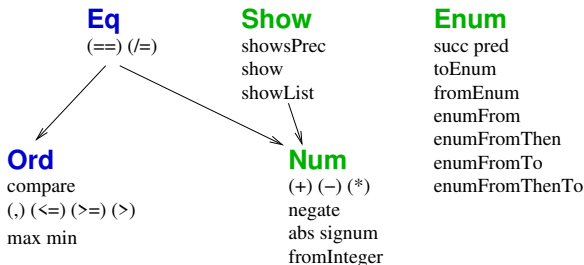
## Die Typklasse `Ord`

- ▶ verlangt von `Instanzen` die Implementierung der Funktionen `compare`, `max` und `min` sowie der Prädikate `(<)`, `(<=)`, `(>=)`, `(>)`.
- ▶ stellt für alle diese Funktionen und Prädikate `Protoimplementierungen` zur Verfügung.

## Minimalvervollständigung bei Instanzbildungen für `Ord`:

- ▶ Implementierung von entweder `compare` oder `(<=)`.

# Typklassen in Haskell bilden eine Hierarchie



Quelle: Fethi Rabhi, Guy Lapalme. [Algorithms - A Functional Approach](#). Addison-Wesley, 1999, Abb. 2.4 (Ausschnitt).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Informelle Beschreibung

...der **Typklasse**:

- ▶ **Eq**: Werte von **Eq**-Typen müssen auf Gleichheit und Ungleichheit vergleichbar sein.
- ▶ **Ord**: Werte von **Ord**-Typen müssen über Gleichheit und Ungleichheit hinaus bezüglich ihrer relativen Größe vergleichbar sein.
- ▶ **Show**: Werte von **Show**-Typen müssen eine Darstellung als Zeichenreihe besitzen.
- ▶ **Num**: Werte von **Num**-Typen müssen mit ausgewählten numerischen Operationen verknüpfbar sein, d.h. addiert, multipliziert, subtrahiert, etc. werden können.
- ▶ **Enum**: Werte von **Enum**-Typen müssen aufzählbar sein, d.h. einen Vorgänger- und Nachfolgerwert innerhalb des Typs besitzen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Beispiel 3: Vordefinierte Typklasse Show

...für Typen, deren Werte als Zeichenreihe dargestellt werden können:

```
type ShowS = String -> String

class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS

  showsPrec _ x s = show x ++ s      -- Protoimplementierungen
  show x = showsPrec 0 x ""
  showList [] = showString "[]"
  showList (x:xs) = showChar '[' . shows x . showl xs
                        where showl []      = showChar ']'
                              showl (x:xs) =
                                showChar ',' . shows x . showl xs
```

Minimalvervollständigung bei Instanzbildungen für Show:

- Implementierung von entweder `show` oder `showsPrec`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9



# Instanzbildungen für die Typklasse Show

...am Beispiel der Typen `Punkt` und `(My)-Bool`:

```
instance Show Punkt where
  show (Pkt (u,v))
    = "(" ++ show u ++ "," ++ show v ++ ") Punkt"
      ++ (if v==0 then " " else " nicht ")
      ++ "auf x-Achse."

show (Pkt (42,4711)) ->> "(42.0,4711.0) Punkt nicht auf x-Achse"
show (Pkt (42,0))    ->> "(42.0,0.0) Punkt auf x-Achse"

newtype MyBool = MB Bool
instance Show MyBool where
  show (MB True)  = "Wahr"
  show (MB False) = "Falsch"

show (MB True)  ->> "Wahr"
show (MB False) ->> "Falsch"
```

**Anmerkung:** `Bool` ist bereits vordefiniert als Instanz von `Show` (`show True ->> "True"`, `show False ->> "False"`) und kann deshalb nicht erneut zur Instanz gemacht werden.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Beispiel 4: Vordefinierte Typklasse Read

...für Typen, deren Werte aus einer Zeichenreihe abgeleitet werden können:

```
type ReadS a = String -> [(a,String)]  
  
class Read a where  
  readsPrec :: Int -> ReadS a  
  readList  :: ReadS [a]  
  readList = ...           -- Protoimpl., hier weggelassen
```

Minimalvervollständigung bei Instanzbildungen für Read:

- Implementierung von `readsPrec`.

...siehe [Standard-Präludium](#) und [Sprachbericht](#) für hier nicht angegebene Hilfsfunktionen von `Read` (und `Show`).

# Beispiel 5: Vordefinierte Typklasse Enum

...für Typen, deren Werte aufgezählt werden können:

```
class Enum a where                                -- Informelle Bedeutung:
  succ, pred      :: a -> a      -- Vorgänger-, Nachfolgerwert
  toEnum          :: Int -> a    -- Typkonversion
  fromEnum        :: a -> Int    -- Typkonversion
  enumFrom        :: a -> [a]    -- [n..]
  enumFromThen    :: a -> a -> [a] -- [n,n'..]
  enumFromTo      :: a -> a -> [a] -- [n..m]
  enumFromThenTo  :: a -> a -> a -> [a] -- [n,n'..m]

  succ            = toEnum . (+1) . fromEnum -- Protoimpl.
  pred            = toEnum . (subtract 1) . fromEnum
  enumFrom x      = map toEnum [fromEnum x ..]
  enumFromTo x    = map toEnum [fromEnum x .. fromEnum y]
  enumFromThen x y =
    map toEnum [fromEnum x, fromEnum y .. fromEnum]
  enumFromThenTo x y z =
    map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

Minimalvervollständigung bei Instanzbildungen für Enum:

- Implementierung von `toEnum` und `fromEnum`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Beispiel 6: Vordefinierte Typklasse Num

...für Typen, deren Werte numerisch behandelt werden können:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a

  x - y      = x + negate y      -- Protoimplemen-
  negate x = 0 - x              -- tierungen
```

Minimalvervollständigung bei Instanzbildungen für Num:

- Implementierung aller Funktionen mit Ausnahme von entweder `negate` oder `(-)`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Weitere numerische Typklassen

...neben bzw. als Erweiterung von `Num` und `Enum`:

```
class (Num a, Ord a)    => Real a where...
class (Real a, Enum a) => Integral a where...
class (Num a)           => Fractional a where...
class (Fractional a)    => Floating a where...
class (Real a, Fractional a) => RealFrac a where...
class (RealFrac a, Floating a) => RealFloat a where...
```

...siehe [Standard-Präludium](#) und [Sprachbericht](#) für Einzelheiten.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

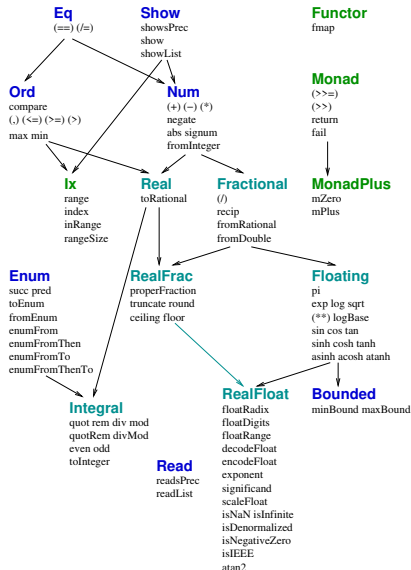
Teil III

Kap. 7

Kap. 8

Kap. 9

# Die result. Typklassenhierarchie im Überblick



Quelle: Fethi Rabhi, Guy Lapalme. *Algorithms - A Functional Approach*.  
Addison-Wesley, 1999, Abb. 2.4.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

406/170

# Beschreibung ausgewählter Typklassen

## Typklasse

- ▶ 'Gleichheit' `Eq`: Klasse der Typen mit Gleichheits- (`==`) und Ungleichheitsrelation (`/=`).
- ▶ 'Ordnung' `Ord`: Klasse der Typen mit Ordnungsrelationen (`<`, `≤`, `>`, `≥`, etc.).
- ▶ 'Numerisch' `Num`: Klasse der Typen, deren Werte sich numerisch verhalten (Bsp.: `Int`, `Integer`, `Float`, `Double`)
- ▶ 'Aufzählung' `Enum`: Klasse der Typen, deren Werte aufgezählt werden können (Bsp.: `[2,4..29] :: Int`).
- ▶ 'Ausgabe' `Show`: Klasse der Typen, deren Werte als Zeichenreihen dargestellt werden können (Bsp.: `Bool`, `Int`, `Integer`, `[Int]`, `(Bool,Int)`,...)
- ▶ 'Eingabe' `Read`: Klasse der Typen, deren Werte aus Zeichenreihen herleitbar sind (Bsp.: `Bool`, `Int`, `Float`, `[Float]`, `[(Bool,Float)]`,...)
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 4.3.3

## Instanzbildung für Typklassen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

**4.3.3**

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9



# Instanzbildung für die Typklasse Eq

...mache Typ `Kurs` zu einem Element von `Eq`:

```
newtype Kurs = K Float
instance Eq Kurs where
    K k1 == K k2 = k1 == k2
```

Beachte: `(==)` ist der in Haskell vordefinierte Vergleich auf Werten vom Typ `Float`:

```
(==) :: Float -> Float -> Bool
```

wohingegen `(==)` der in der Instanzdeklaration selbstdefinierte Vergleich auf Werten vom Typ `Kurs` ist:

```
(==) :: Kurs -> Kurs -> Bool
```

dessen Definition sich auf den Vergleich `(==)` abstützt.

# Anwendungen von Kurs als Eq-Typ

...mit der Eq-Instanzbildung für `Kurs` unmittelbar möglich:

```
gs_kurs1      == gs_kurs1      ->> True
```

```
gs_kurs1      /= gs_kurs2      ->> True
```

```
gs_ausschlag1 == gs_ausschlag1 ->> True
```

```
gs_ausschlag1 /= gs_ausschlag2 ->> True
```

```
gs_verlauf1   == gs_verlauf1   ->> True
```

```
gs_verlauf1   /= gs_verlauf2   ->> True
```

# Instanzbildung für die Typklasse Ord

...mache Typ `Kurs` zu einem Element von `Ord`:

```
newtype Kurs = K Float
instance Ord Kurs where
    K k1 <= K k2 = k1 <= k2
```

**Beachte:** `(<=)` ist der in Haskell vordefinierte Vergleich auf Werten vom Typ `Float`:

```
(<=) :: Float -> Float -> Bool
```

wohingegen `(<=)` der in der Instanzdeklaration selbstdefinierte Vergleich auf Werten vom Typ `Kurs` ist:

```
(<=) :: Kurs -> Kurs -> Bool
```

dessen Definition sich auf den Vergleich `(<=)` abstützt.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

411/170

# Anwendungen von Kurs als Ord-Typ

...mit der `Ord`-Instanzbildung für `Kurs` unmittelbar möglich:

```
gs_kurs1 <= gs_kurs1 ->> True
```

```
gs_kurs1 > gs_kurs2 ->> False
```

```
gs_ausschlag1 >= gs_ausschlag2 ->> False
```

```
gs_ausschlag1 'max' gs_ausschlag2 ->> (K 7.2, K 9.7)
```

```
gs_verlauf1 'min' gs_verlauf2
```

```
->> [(K 2.4, K 7.9), (K 3.2, K 5.7), (K 2.8, K 9.3)]
```

```
gs_verlauf1 'compare' gs_verlauf2 ->> GT
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Übungsaufgabe 4.3.3.1

Mache die neuen Typen

- ▶ `Pegelstand`
- ▶ `Positionsdaten`

aus Kapitel 4.2.2 zu Instanzen der Typklassen

1. `Eq` auf zwei alternative, gleichwertige Weisen durch  
Definition von:
  - 1.1 `(==)`
  - 1.2 `(/=)`
2. `Ord` auf zwei alternative, gleichwertige Weisen durch  
Definition von:
  - 2.1 `(<=)`
  - 2.2 `compare`

# Kapitel 4.3.4

## Automatische Instanzbildung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

**4.3.4**

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Automatische Instanzbildungen

Für ausgewählte Typklassen können explizite Instanzbildungen wie hier für `Pegelstand` und `Koordinate` für Typklasse `Eq`:

```
newtype Pegelstand = Pgl Float
```

```
instance Eq Pegelstand where
```

```
  Pgl p1 == Pgl p2 = p1 == p2
```

```
newtype Koordinate = Koordinate Float
```

```
instance Eq Koordinate where
```

```
  Koordinate k1 /= Koordinate k2 = k1 /= k2
```

bedeutungsgleich durch automatische Instanzbildungen mittels einer `deriving`-Klausel ersetzt werden:

```
newtype Pegelstand = Pgl Float deriving Eq
```

```
newtype Koordinate = Koordinate Float deriving Eq
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

415/170

# Automatische Instanzbildungen

...sind möglich (ausschließlich!) für die vordefinierten Typklassen `Eq`, `Ord`, `Enum`, `Bounded`, `Show` und `Read` (s. [Kap. 11.4.3](#)).

Für unser durchgehendes Beispiel liefern die automatischen Instanzbildungen das gewünschte Verhalten und sind deshalb am bequemsten:

```
newtype Kurs          = K Float
                        deriving (Eq,Ord,Show)

newtype Pegelstand = Pgl Float
                        deriving (Eq,Ord,Show)

newtype Koordinate = Koordinate Float
                        deriving (Eq,Ord,Show)
```

Ist ein anderes als das automatisch erzeugte ‘*offensichtliche*’ Verhalten gewünscht, muss die Instanzbildung explizit mit einer `instance`-Deklaration erfolgen (s. [Kap. 11.4.3](#)).



# Beispiel anhand des Typs Punkt (1)

...die Deklaration mit **deriving**-Klausel:

```
type X_Koord = Float
type Y_Koord = Float
newtype Punkt = Pkt (X_Koord,Y_Koord) deriving Eq
```

ist äquivalent zur **Instanz**-Deklaration:

```
instance Eq Punkt where
  (Pkt (u,v)) == (Pkt (u',v')) = (u==u') && (v==v')
```

$\underbrace{\hspace{10em}}_{(==) \text{ auf Punkt}} \qquad \underbrace{\hspace{10em}}_{(==) \text{ auf Float}}$

Ist 'Gleichheit' von Punkten anders gewünscht, etwa Gleichheit bereits bei übereinstimmender x- oder y-Koordinate, so ist eine **Instanz**-Deklaration unumgänglich:

```
instance Eq Punkt where
  (Pkt (u,v)) == (Pkt (u',v')) = (u==u') || (v==v')
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Beispiel anhand des Typs Punkt (2)

...oder bei:

‘Gleichheit’ als ‘nicht zu weit voneinander entfernt’, d.h. innerhalb eines Quadrats mit Seitenlänge  $\varepsilon > 0$  umeinander:

```
instance Eq Punkt where
  (Pkt (u,v)) == (Pkt (u',v'))
    = (abs (u-u') <= epsilon/2) &&
      (abs (v-v') <= epsilon/2) where epsilon = 0.1
```

‘Gleichheit’ als ‘gleich weit vom Ursprung (0,0) entfernt’, d.h. auf derselben Kreislinie um den Ursprung:

```
instance Eq Punkt where
  (Pkt (u,v)) == (Pkt (u',v'))
    = sqrt (u*u + v*v) == sqrt (u'*u' + v'*v')
```

oder...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Antibeispiel anhand des Typs Punkt (1)

Beachte: Folgender Instanzbildungsversuch schläge fehl:

```
newtype Temperatur = Tmp Float
newtype Regenmenge = Rmg Float

instance Eq Punkt where
  (Pkt (u,v)) == (Pkt (u',v'))
    = (Tmp u == Tmp u') || (Rmg v == Rmg v')
    
$$\underbrace{((=)) \text{ auf Temperatur, nicht auf Float}}_{\text{}} \quad || \quad \underbrace{((=)) \text{ auf Regenmenge, nicht auf Float}}_{\text{}}$$

```

...da **Temperatur** und **Regenmenge** selbst noch keine Instanzen von **Eq** sind und das Relationssymbol **(==)** deshalb für **Temperatur**- und **Regenmenge**-Werte noch keine Bedeutung hat (abgesehen vom zweifelhaften semantischen Zusammenhang von Orts-, Temperatur- und Regenmengendaten).

# Antibeispiel anhand des Typs Punkt (2)

**Beachte:** Folgende Instanzbildung wäre **syntaktisch möglich**, aber **semantisch kontraintuitiv**: Punkte würden durch diese Instanzbildung als

- ▶ gleich angesehen, wenn sie sich in x- oder/und y-Koordinate unterscheiden.
- ▶ ungleich angesehen, wenn sie in x- und y-Koordinate übereinstimmen.

```
instance Eq Punkt where
```

```
(Pkt (u,v)) == (Pkt (u',v')) = (u/=u') || (v/=v')
```

```
(Pkt (u,v)) /= (Pkt (u',v')) = (u==u') && (v==v')
```

...in der Anwendung wäre diese kontraintuitive Festlegung der Bedeutung der Relatorsymbole (**==**) und (**/=**) für Punktwerte **Quelle vielfacher Programmierfehler** und daher **abzulehnen**.

# Kapitel 4.3.5

## Selbstdefinierte Typklassen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

**4.3.5**

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Selbstdefinierte Typklassen, Wiederverwendung

...auf Wertpapier-, Wasserstands- und Positionsdaten sind Operationen zur Auswertung einzelner Ereignisse und Ereignisfolgen anwendbar.

Alle diese Funktionen haben ähnliche Funktionalität und ähnliche Namen:

- ▶ `auswertung`, `auswertung'`, `auswertung''`
- ▶ `reihenausw`, `reihenausw'`, `reihenausw''`

Es bietet sich deshalb an,

- ▶ diese Operationen in einer selbstdefinierten neuen Typklasse zu bündeln,
- ▶ so Namen einzusparen und wiederzuverwenden,
- ▶ die Verwendungsähnlichkeit der Typen für Wertpapier-, Wasserstands- und Positionsdaten auszudrücken.

# Bsp. 1: Selbstdef. Typklasse Analysierbar

Wir bündeln die Auswertungsfunktionen auf Wertpapier-, Wasserstands- und Positionsdaten in der selbstdefinierten neuen Typklasse `Analysierbar`:

```
class (Ord a, Fractional a) => Analysierbar a where
  auswertung :: (a,a) -> (a,a,a)
  reihenausw :: [(a,a)] -> [(a,a,a)]
  geglaettet :: a -> a -> a

  reihenausw [] = []           -- Protoimplementierung
  reihenausw ((x,y) : xys)    -- für reihenausw
    = (auswertung (x,y)) : reihenausw xys
  geglaettet x y = (x+y)/2    -- Vollst. Implementierung
                                -- für geglaettet
```

Minimalvervollst. bei Instanzbildungen für `Analysierbar`:

- Implementierung von `auswertung`

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Vorbereitungen

Die Instanzbildungen für `Kurs`, `Pegelstand` und `Koordinate`:

```
newtype Kurs          = K Float
                        deriving (Eq,Ord,Show)
newtype Pegelstand    = Pgl Float
                        deriving (Eq,Ord,Show)
newtype Koordinate    = Koordinate Float
                        deriving (Eq,Ord,Show)
```

für die Typklasse `Analysierbar` setzen folgende (hier als `Übungsaufgabe` gelassene) nicht automatisch ableitbare `Instanzbildungen` für die Typklasse `Num` voraus (Typ `Float` ist vordefinierte Instanz von `Num`):

```
instance Num Kurs where... -- Vervollständigung:
instance Num Pegelstand where... -- Übungsaufgabe
instance Num Koordinate where...
```



# Instanzbildungen für Typklasse Analysierbar

...für die Typen **Float**, **Kurs**, **Pegelstand** und **Koordinate**:

```
instance Analysierbar Float where
  auswertung (f1,f2) = (f1,f2,geglaettet f1 f2)

instance Analysierbar Kurs where
  auswertung (K k1,K k2)
    = (K k1,K k2,K (geglaettet k1 k2))
  geglaettet k k' = (4*k+6*k')/10      -- überschrieben!

instance Analysierbar Pegelstand where
  auswertung (Pgl p1,Pgl p2)
    = (Pgl p1,Pgl p2,Pgl (geglaettet p1 p2))

instance Analysierbar Koordinate where
  auswertung (Koordinate k1,Koordinate k2)
    = (Koordinate k1,Koordinate k2,
        Koordinate (geglaettet k1 k2))
  geglaettet k k' = k+k'              -- überschrieben!
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

## Bsp. 2: Selbstdefinierte Typklasse Warnung

...die Auswertung von Wertpapier- und Wasserstandsdaten mag vorteilhafte oder gefährliche Situationen erkennen lassen, die entsprechende 'Warnungen' ermöglichen sollte.

Wir bündeln diese Funktionen in einer weiteren neuen Typklasse Warnung, die sich auf Analysierbar abstützt:

```
class (Analysierbar a) => Warnung a where
  warnung    :: (a,a) -> String
  warnreihe  :: [(a,a)] -> String
  warnreihe xys = warnung (wr xys (0,0)) -- Proto-
    where wr [] pq = pq                  -- implementierung
          wr ((x,y) : xys) (p,q) = wr xys (x+p,y+q)
```

Minimalvervollständigung bei Instanzbildungen für Warnung:

- Implementierung von warnung

# Instanzbildungen für Typklasse Warnung (1)

...für Typ `Kurs`:

```
instance Warnung Kurs where
  warnung (K k1, K k2)
    | k2 > 9*k1 = "Verkaufen! Aktie zu spekulativ."
    | k2 > 6*k1 = "Halten! Aktie an Spekulationsschwelle."
    | k2 > 3*k1 = "Zukaufen! Aktie hat Phantasie."
    | otherwise = "Verkaufen! Aktie ohne Phantasie."

-- Für Kurs passt uns die Protoimplementierung von
-- warnreihe, so dass nichts weiter für uns zu tun ist.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Instanzbildungen für Typklasse Warnung (2)

...für Typ `Pegelstand`:

```
instance Warnung Pegelstand where
  warnung (Pgl p1,Pgl p2)
    | p2 >= 100 = "Evakuieren! Deich kurz vor Bruch."
    | p2 >= 80  = "Achtung! Deich an Belastungsgrenze."
    | p1 <= 20  = "Sperrwerk öffnen! Pegel zu niedrig."
    | otherwise = "Pegel im Normalbereich."

-- Für Pegelstand passt uns die Protoimplementierung von
-- warnreihe nicht. Wir überschreiben sie daher:
warnreihe pgs = meldung anteil
  where anzahlEreignisse = length pgs
        anzahlGefahrEreignisse
          = length [max | (Pgl min,Pgl max) <- pgs, max >= 100]
        anteil = (anzahlGefahrEreignisse * 100)
                  'div' anzahlEreignisse

meldung n
  | n >= 30  = "Anteil Gefahrereignisse hoch"
  | n >= 10  = "Anteil Gefahrereignisse moderat"
  | otherwise = "Anteil Gefahrereignisse gering"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Instanzbildungen für Typklasse Warnung (3)

...für Typ `Koordinate`.

Für `Positionsdaten` als Punkte im zwei- bzw. dreidimensionalen mathematischen Raum besteht

- ▶ kein Grund

Warnungen auszugeben. Deshalb ist eine Instanzbildung von `Koordinate` für die Typklasse `Warnung` unnötig und wird

- ▶ unterlassen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Vordefinierte Instanzen von Typklassen

Viele Typen, insbesondere

- ▶ Elementartypen (`Bool`, `Char`, `Int`,...)
- ▶ Tupel von Elementartypen
- ▶ Listen von Elementartypen (speziell Zeichenreihen)
- ▶ ...

sind bereits **vordefinierte Instanzen** der passenden Typklassen.

Deshalb sind wir damit ausgekommen, Instanzbildungen für

- ▶ `Kurs`, `Pegelstand` und `Koordinate` vorzunehmen.

Auf die darauf aufbauenden Tupel- und Listentypen wie `Kursausschlag`, `Kursverlauf`, etc., haben sich die Eigenschaften automatisch übertragen.

# Typsicherheit bleibt gewahrt!

Beachte: Die Funktionen

- ▶ `auswertung`, `reihenausw` der Typklasse `Analysierbar` sind `überladen` und auf
  - ▶ `Wertpapier-`, `Wasserstands-` und `Positionsdaten` anwendbar.
- ▶ `warnung`, `warnreihe` der Typklasse `Warnung` sind `überladen` und auf
  - ▶ `Wertpapier-` und `Wasserstandsdaten` anwendbar.

Dabei gilt: Typsicherheit wird `nicht` korrumpiert:

- ▶ Alle Aufrufe der überladenen Funktionen erfolgen (wg. der jeweiligen Instanzbildungen) mit `typspezifischem Code`!

Die `Typsicherheit` bleibt deshalb `in vollem Umfang gewahrt`.

# Anwendungsszenario für selbstdef. Typklassen

...statt Standardoperatoren oder -relatoren mit einer Nicht-standardbedeutung zu unterlegen wie in:

```
instance Eq Punkt where
  (Pkt (u,v)) == (Pkt (u',v'))
    = sqrt (u*u +v*v) == sqrt (u'*u' + v'*v')
```

besser eine neue Typklasse einführen mit sprechend(er)en Namen:

```
class Aehnlich a where
  aehnlich :: a -> a -> Bool

instance Aehnlich Punkt where
  aehnlich (Pkt (u,v)) (Pkt (u',v'))
    = sqrt (u*u +v*v) == sqrt (u'*u' + v'*v')
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9



# Anwendungsszenario für selbstdef. Typklassen

...eingebettet in einen größeren Kontext:

```
type X_Koord  = Float
type Y_Koord  = Float
newtype Punkt = Pkt (X_Koord,Y_Koord) deriving Eq

class Aehnlich a where                -- Jeder Typ kann Instanz
    aehnlich :: a -> a -> Bool        -- von Aehnlich werden.

instance Aehnlich Punkt where
    aehnlich (Pkt (u,v)) (Pkt (u',v'))
        = sqrt (u*u +v*v) == sqrt (u'*u' + v'*v')

class (Eq a) => Aehnlich' a where     -- Nur Eq-Instanzen können
    aehnlich' :: a -> a -> Bool      -- Inst. v. Aehnlich' werden.

instance Aehnlich' Punkt where
    aehnlich' (Pkt (u,v)) (Pkt (u',v')) = (u==u') || (v==v')
```

...wobei für `(==)`, `(/=)` die 'erwartete' Standardbedeutung für `Punkt`-Werte bewahrt wird, zugew. d. die `deriving`-Klausel.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 4.3.6

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

**4.3.6**

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Haskells Philosophie zu Typen und Typklassen

...als **Faustregel** formuliert: Bei **Einführung** eines **neuen Typs**:

- ▶ Überlege, welche Operationen und Relationen auf Werte dieses Typs anwendbar sein sollen und möglicherweise in bereits existierenden Typklassen konzeptuell enthalten sind.
- ▶ Mache den neuen Typ zu einer Instanz all derjenigen Typklassen, zu denen diese Operationen und Relationen gehören; oft reicht dafür eine **deriving**-Klausel aus.
- ▶ Sind auf die Werte des neuen Typs Operationen und Relationen anwendbar, die noch nicht in dieser oder vergleichbarer Form in einer Typklasse gebündelt sind, so
  - ▶ führe eine **neue Typklasse** mit diesen Operationen und Relationen ein; ggf. zusammen mit vollständigen Implementierungen oder zu vervollständigenden Protoimplementierungen, wo passend

wenn zu erwarten ist, dass sie konzeptuell auch für weitere erst noch zu definierende Datentypen relevant sind.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Typklassen vs. objektorientierte Klassen

Haskells Typklassenkonzept unterscheidet sich wesentlich vom Klassenkonzept objektorientierter Sprachen.

## Objektorientiert: Klassen

- ▶ dienen der Strukturierung von Programmen.
- ▶ liefern Blaupausen zur Generierung von Werten.

## In Haskell: Typklassen

- ▶ sind Sammlungen von Typen, deren Werte in 'ähnlicher' Weise verarbeitet werden können.
- ▶ erhalten Typen als Elemente explizit durch Instanzbildung oder implizit durch automatische Instanzbildung (**deriving**-Klausel) zugewiesen.
- ▶ dienen **nicht** der Strukturierung von Programmen; liefern **keine** Blaupausen zur Generierung von Werten.

# Ausblick

...wir kommen auf

- ▶ Typdefinitionen
- ▶ Typklassen

im Zusammenhang mit

- ▶ algebraischen Datentypdeklarationen (Kap. 5)
- ▶ *Ad hoc* Polymorphie, Überladung (Kap. 11)

noch einmal zurück.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

4.3.6

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Übungsaufgabe 4.3.6.1

Vergleiche das **Typklassenkonzept** aus **Haskell** mit dem **Schnittstellenkonzept** aus **Java**.

Welche Gemeinsamkeiten, welche Unterschiede fallen auf?

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.3.5

**4.3.6**

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 4.4

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

**4.4**

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8


Kap. 9


Teil IV


Kap. 10

Kap. 11

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 4 (1)

 Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 7.1, Typsynonyme mit type; Kapitel 7.2, Einfache algebraische Typen mit data und newtype; Kapitel 7.4, Automatische Instanzen von Typklassen; Kapitel 7.8, Eigene Klassen definieren)

 Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 2, Expressions, types and values; Kapitel 3, Numbers)

 Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3, Types and classes; Kapitel 8, Declaring types and classes)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

440/170



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 4 (2)



Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 2, Believe the Type)



Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 3, Defining Types, Streamlining Functions; Kapitel 6, Using Typeclasses)



Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 13.4, A tour of the built-in Haskell classes)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

441/170

# Kapitel 5

## Datentypdeklarationen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

**Kap. 5**

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

# Kapitel 5.1

## Überblick, Orientierung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

# Grundlegende Datentypstrukturen

...in Programmiersprachen sind:

- ▶ Aufzählungstypen
- ▶ Produkttypen
- ▶ Summentypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

# Charakterisierung und typische Beispiele

## Aufzählungstypen

- Typen mit jeweils endlich vielen Werten.

Beispiel: Typ Jahreszeiten mit Werten Fruehling, Sommer, Herbst und Winter.

## Produkttypen (oder Verbundtypen (engl. record types))

- Typen mit möglicherweise unendlich vielen (Tupel-) Werten.

Beispiel: Typ Mensch mit Werten (Adam,Riese,männlich), (Ada,Lovelace,weiblich), etc.

## Summentypen (oder Vereinigungstypen)

- Typen mit Werten, die sich aus der Vereinigung der Werte verschiedener Typen mit jeweils möglicherweise unendlich vielen Werten ergeben.

Beispiel: Typ Sammelurium als Vereinigung der (Werte der) Typen Buch, KFZ, Haustier, etc.

# Bereits besprochen: Typsynonyme, neue Typen

...mittels:

- **type**-Deklarationen zur Definition von Typsynonymen, d.h. neue Namen für existierende Typen, Typalias:

```
type Kurs = Float           type Pegelstand = Float           type Koordinate = Float
type Niedrigst = Kurs       type Niedrig = Pegelstand        type X = Koordinate
type Hoechst = Kurs        type Hoch = Pegelstand           type Y = Koordinate
type Kursauschlag          type Messung = (Niedrig,Hoch)      type Ebenenpunkt = (X,Y)
    = (Niedrigst,Hoechst)
```

...keine zusätzliche Typsicherheit; unterstützen Transparenz.

- **newtype**-Deklarationen zur Definition von Typidentitäten:

```
newtype Kurs = K Float     newtype Pegelstand = Pgl Float    newtype Koordinate = Koordinate Float
type Niedrigst = Kurs       type Niedrig = Pegelstand        type X = Koordinate
type Hoechst = Kurs        type Hoch = Pegelstand           type Y = Koordinate
type Kursauschlag         type Messung = (Niedrig,Hoch)      type Ebenenpunkt = (X,Y)
    = (Niedrigst,Hoechst)
```

...Typsicherheit durch (Datenwert-) Konstruktoren.

Allerdings: Beschränkt auf 1 Konstruktor mit 1 (Daten-) Feld.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

446/170

# Neu: Algebraische Datentypen

...mittels:

- ▶ **data**-Deklarationen zur Definition **originär neuer Typen**, von **Aufzählungs-, Produkt- und Summentypen**.

- ▶ **Aufzählungstypen**

```
data Jahreszeiten = Fruehling | Sommer  
                  | Herbst | Winter
```

```
data Geschlecht  = Maennlich | Weiblich
```

- ▶ **Produkttypen**

```
type Vorname  = String
```

```
type Nachname = String
```

```
data Mensch   = M Vorname Nachname Geschlecht
```

- ▶ **Summentypen**

```
data Baum = Blatt Int | Wurzel Baum Int Baum
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

447/170

# Damit

...Datentypdeklarationen in **Haskell** mittels dreier **Sprachkonstrukte**:

- ▶ **type** (Kap. 4.1)
- ▶ **newtype** (Kap. 4.2)
- ▶ **data** (Kap. 5.2)

Anschließend:

- ▶ Funktionen auf algebraischen Datentypen (Kap. 5.3)
- ▶ Feldsyntax für algebraische Datentypen (Kap. 5.4)
- ▶ Anwendungshinweise (Kap. 5.5)
- ▶ Literaturhinweise (Kap. 5.6)



# Kapitel 5.2

## Algebraische Datentypen (data)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

**5.2**

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

# Kapitel 5.2.1

## Aufzählungstypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

**5.2.1**

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

# Beispiele vordefinierter Aufzählungstypen

Typ der Ordnungswerte, 3 Werte:

```
data Ordering = LT | EQ | GT deriving (Eq,Ord,  
                                         Bounded,  
                                         Enum, Read,  
                                         Show)
```

Typ der Wahrheitswerte, 2 Werte:

```
data Bool = False | True deriving (Eq,Ord,Bounded,  
                                    Enum,Read,Show)
```

Trivialer Typ (oder Nulltupeltyp), 1 Wert:

```
data () = () deriving (Eq,Ord,Bounded,Enum,Read,  
                       Show)
```

...Nulltupeltyp und einziger (def.) Wert ident bezeichnet: `()`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

451/1700

# Beispiele selbstdefinierter Aufzählungstypen

```
data Jahreszeiten = Fruehling | Sommer  
                  | Herbst | Winter  
    deriving (Eq,Ord,Bounded,  
              Enum,Read,Show)
```

```
data Spielfarbe   = Karo | Herz | Pik | Kreuz  
    deriving (Eq,Ord,Bounded,  
              Enum,Read,Show)
```

```
data Werkzeuge    = Montag | Dienstag | Mittwoch  
                  | Donnerstag | Freitag  
    deriving (Eq,Ord,Bounded,  
              Enum,Read,Show)
```

```
data Wochenende   = Samstag | Sonntag  
    deriving (Eq,Ord,Bounded,  
              Enum,Read,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

452/170

# Kapitel 5.2.2

## Produkttypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

**5.2.2**

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

# Beispiele selbstdefinierter Produkttypen

```
type Vorname      = String           -- Personendaten
type Nachname     = String
data Geschlecht   = Maennlich
                  | Weiblich deriving (Eq, Show)
type Gemeinde     = String           -- Adressdaten
type Strasse      = String
type Hausnr       = Int
type Land         = String

data Person       = P Vorname Nachname Geschlecht d...
data Anschrift    = A Gemeinde Strasse Hausnr Land d...
data Einwohner    = E Land Gemeinde [Person] deriving...
data Wohnsitze    = W Land (Person -> [Anschrift])
data Gemeldet     = G (Land -> Gemeinde -> Strasse
                      -> Hausnr -> [Person])
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

454/170

# Kapitel 5.2.3

## Summentypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

**5.2.3**

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

# Beispiele vordefinierter Summentypen

## Listen

```
data [a] = []  
         | a : [a] deriving (Eq,Ord)  
           -- Kein gültiges Haskell;  
           -- nur zur Illustration!
```

## Der Möglicherweise-Typ

```
data Maybe a = Nothing  
             | Just a deriving (Eq,Ord,Read,Show)
```

## Der Entweder/Oder-Typ

```
data Either a b = Left a  
                | Right b deriving (Eq,Ord,Read,  
                                   Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

456/170



# Beispiele selbstdefinierter Summentypen (1)

```
type Autor          = String          -- Buch-/E-Buchdaten
type Titel          = String
type Verlag         = String
type Auflage        = Int
type Lieferbar      = Bool
type LizenzBisJahr  = Int
type Hauptdarsteller = [String]      -- Filmdaten
type Regisseur      = String
type Sprachen       = [String]
type Kuenstler      = String          -- Musikaufnahmedaten
type Std            = Int
type Min            = Int
type Sek            = Int
type Spieldauer     = (Std,Min,Sek)

data BildSchriftUndTontraeger =
    Buch Autor Titel Verlag Auflage Lieferbar
  | E_Buch Autor Titel Verlag LizenzBisJahr
  | DVD Titel Hauptdarsteller Regisseur Sprachen
  | CD Kuenstler Titel Spieldauer deriving (Eq,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

# Beispiele selbstdefinierter Summentypen (2)

```
type Autor          = String          -- Buch-/E-Buchdaten
type Titel          = String
type Verlag         = String
type Auflage        = Int
type Lieferbar      = Bool
data Kategorie      = PKW | LKW | Bus | Cabrio | SUV
                    deriving (Eq,Show)  -- Fahrzeugdaten

type Marke          = String
type Listenpreis    = Float
data Tierart        = Hund | Katze | Maus | Kanarienvogel
                    deriving (Eq,Show)  -- Haustierdaten

type Rufname        = String
type Gewicht_in_kg  = Float
type Vielfrass      = Bool

data Sammelurium =
    Buch Autor Titel Verlag Auflage Lieferbar
    | KFZ Kategorie Marke Listenpreis
    | Haustier Tierart Rufname Gewicht_in_kg Vielfrass
    deriving (Eq,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

458/170

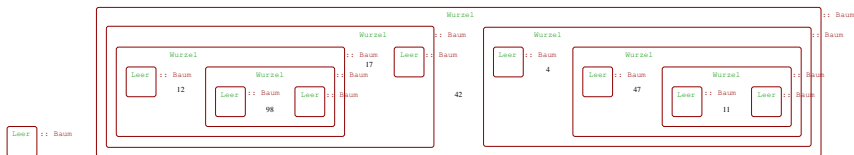
# Beispiele rekursiver Summentypen (1)

## Zweistellige Bäume (oder Binärbäume)

```
data Baum = Leer
          | Wurzel Baum Int Baum
          deriving (Eq,Ord,Show)
```

## Veranschaulichung: Werte vom Typ Baum:

Werte vom Typ  
Baum



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

459/170

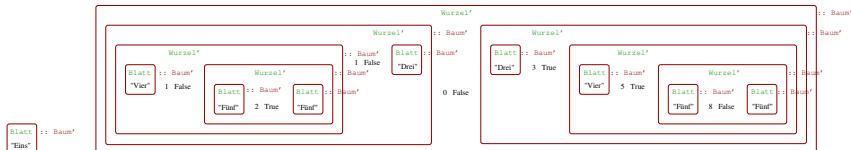
# Beispiele rekursiver Summentypen (2)

## Zweistellige Bäume (oder Binärbäume)

```
data Baum' = Blatt String
           | Wurzel' Baum' Int Bool Baum'
           deriving (Eq, Ord, Show)
```

## Veranschaulichung: Werte vom Typ Baum':

Werte vom Typ  
Baum'



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

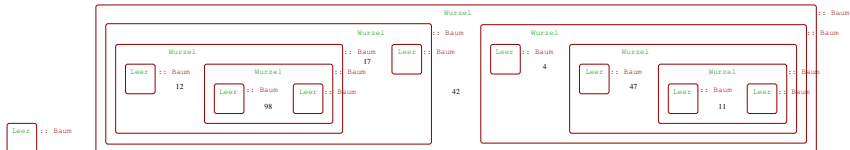
Kap. 8

460/1700

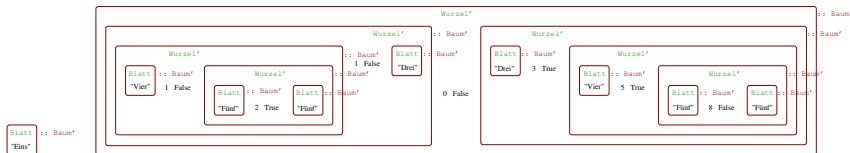
# Beispiele rekursiver Summentypen (3)

Werte der Typen **Baum** und **Baum'**, zum Vergleich auf einer Seite:

Werte vom Typ  
**Baum**



Werte vom Typ  
**Baum'**



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

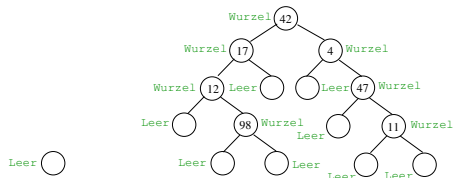
461/1700

# Beispiele rekursiver Summentypen (4)

## Werte d. Typen **Baum** und **Baum'**: 'Konventionelle' Darstellung

Zwei Werte vom Typ

**Baum**



data **Baum**

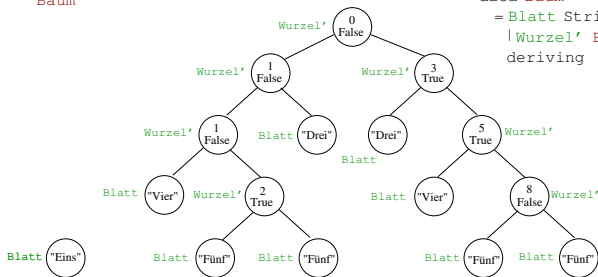
= **Leer**

| **Wurzel** **Baum** Int **Baum**

deriving (Eq, Ord, Show)

Zwei Werte vom Typ

**Baum'**



data **Baum'**

= **Blatt** String

| **Wurzel'** **Baum'** Int Bool **Baum'**

deriving (Eq, Ord, Show)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

**5.2.3**

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

# Beispiele rekursiver Summentypen (5)

## Dreistellige Bäume (oder Trinärbäume)

```
data Tbaum = Nichts
           | Gabel Person Tbaum Tbaum Tbaum
           deriving (Eq, Ord, Show)
```

```
data Tbaum' = Laub Person [Anschrift]
            | Gabel' Person [Anschrift]
              Tbaum' Tbaum' Tbaum'
            deriving (Eq, Ord, Show)
```

## *n*-stellige Bäume

```
data Nbaum      = NB Int [Nbaum]
                deriving (Eq, Ord, Show)

data Nbaum'     = NB' (Person, [Anschrift]) [Nbaum']
                deriving (Eq, Ord, Show)
```

```
data Nadelbaum = Nb String Int Char Baum Tbaum
               [Nadelbaum] deriving...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

463/170

# Beispiele rekursiver Summentypen (6)

## Suchbäume

```
type Schluessel  = Int
type Information = String
data Suchbaum    = Sb Schluessel Information
                  | Sk Schluessel Information
                  | Suchbaum Suchbaum
                  deriving (Eq,Ord,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

464/170



# Beispiele rekursiver Summentypen (7)

## Kartei

```
type Soz_Vers_Nr = Int
type Schluessel  = Soz_Vers_Nr
type Str         = Strasse
type Hnr         = Hausnr
type Info        = (Person,
                    (Land -> Gemeinde -> [(Str,Hnr)]))

data Kartei = Kb Schluessel Info
           | Kk Schluessel Info Kartei Kartei
           deriving (Eq,Ord,Show)

-- Kb für Karteiblatt.
-- Kk für Karteikasten.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

465/170

# Beispiele wechselw. rekursiver Summentypen

## Bürger Netzwerk von Freunden und verwandten und bekannten Nachbarn

```
type Verwandt = Bool
type Bekannt  = Bool
data Wohnform = EFH | ZFH | MFH | DH | RH | HH | PH
               deriving (Eq,Show)

data Buerger   = B Person Wohnform Nachbarn Freunde
               deriving (Eq,Show)

data Nachbarn = N [(Buerger,Verwandt,Bekannt)]
               deriving (Eq,Show)

data Freunde  = F [Buerger]
               deriving (Eq,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

466/170

# Kapitel 5.2.4

## Allgemeines Muster

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

**5.2.4**

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

# Allgemeines Muster

...algebraischer Datentypdefinitionen:

```
data Typename = Con_1 t_11 ... t_1k_1
               | Con_2 t_21 ... t_2k_2
               ...
               | Con_n t_n1 ... t_nk_n
```

Sprechweisen:

- ▶ **Typename**: Freigewählter frischer Typname
- ▶ **Con\_i**: Freigewählte frische (Datenwert-) Konstruktor-namen
- ▶ **k\_i**: Stelligkeit des Konstruktors **Con\_i**
- ▶ **t\_ij**: Namen existierender Typen

Beachte: **Typname** und **Konstruktoren** müssen stets mit einem Großbuchstaben beginnen (siehe z.B. **Bool**, **True**, **False**)!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

468/170

# (Datenwert-) Konstruktoren

...können als Funktionsdefinitionen gelesen werden:

```
Con_i t_i1 ... t_ik_i -> Typename
```

Die Konstruktion von Werten eines algebraischen Datentyps erfolgt durch Anwendung eines Konstruktors auf Werte 'passenden' Typs:

```
v_i1 :: t_i1 ... v_ik_i :: t_ik_i  
Con_i v_i1 ... v_ik_i :: Typename
```

Beispiele:

```
P "Adam" "Riese" Maennlich :: Person  
A "Wien" "Karlsplatz" 13 "Austria" :: Anschrift  
E_Buch "Simon Thompson" "Haskell" "Pearson" 2018  
      :: BildSchriftUndTontraeger  
Haustier Katze "Garfield" 3.14 True :: Sammelurium
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

469/170

# Kapitel 5.2.5

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

**5.2.5**

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

# Art u. Anzahl der (Datenwert-) Konstruktoren

...**algebraischer Datentypen** liefern in **Haskell** den Schlüssel zu

- ▶ Aufzählungstypen
- ▶ Produkttypen
- ▶ Summentypen

**Hinweis:** Die Bezeichnungen **Produkt-** und **Summentyp** sind Grund in der Gesamtheit von **algebraischen Datentypen** zu sprechen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

471/170

# Kategorisierung

**Aufzählungstypen** gekennzeichnet durch

- ▶ ausschließlich nullstellige Konstruktoren.

**Produkttypen** gekennzeichnet durch

- ▶ genau einen Konstruktor, der nicht nullstellig ist.

**Summentypen** gekennzeichnet durch

- ▶ mehrere Konstruktoren, von denen mindestens einer nicht nullstellig ist.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

472/170



# Aufzählungstypen

...gekennzeichnet durch **ausschließlich 0-stellige Konstruktoren**:

Beispiele:

```
data ()          = ()          -- Ein 0-st. Kon.
data Bool        = False | True -- Zwei 0-st. Kon.
data Ordering    = LT | EQ | GT -- Drei 0-st. Kon.
data Spielfarbe  = Karo | Herz
                  | Pik | Kreuz -- Vier 0-st. Kon.
data Werktag     = Montag | Dienstag
                  | Mittwoch | Donnerstag
                  | Freitag    -- Fünf 0-st. Kon.
```

Wertbeispiele:

- ▶ `()` einziger (def.) Wert des Typs `()`.
- ▶ `False`, `True` einzige (def.) Werte des Typs `Bool`.
- ▶ `LT`, `EQ`, `GT` einzige (def.) Werte des Typs `Ordering`.
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

473/1700

# Produkttypen

...gekennzeichnet durch **genau einen nicht 0-stelligen Konstruktor**:

Beispiele:

```
data Person      = P Vorname Nachname Geschlecht
data Anschrift   = A Gemeinde Strasse Hausnr Land
data Einwohner   = E Land Gemeinde [Person]
data Wohnsitze   = W Land (Person -> [Anschrift])
data Gemeldet    = G (Land -> Gemeinde -> Strasse
                    -> Hausnr -> [Person])
```

Wertbeispiele:

```
adam = P "Adam" "Riese" Maennlich  :: Person
ada  = P "Ada" "Lovelace" Weiblich  :: Person
E "Austria" "Wien" [adam,ada]      :: Einwohner
W "Australia" ws :: Wohnsitze
ws = \p -> case p of
    adam -> [A "Perth" "Main St" 42 "Australia"]
    ada  -> [A "Sydney" "High St" 1 "Australia",
             A "Adelaide" "1st Ave" 10 "Australia"]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

474/170

# Summentypen

...gekennzeichnet durch **mehrere Konstruktoren**, davon mindestens einer nicht 0-stellig:

Beispiele:

```
data [a]          = []          -- Kein gültiges Haskell;  
                  | a : [a]      -- nur zur Illustration!  
  
data Maybe a      = Nothing  
                  | Just a  
  
data Either a b   = Left a  
                  | Right b
```

Wertbeispiele:

```
[] :: []; [1,2,3] :: [Int]; [True,False,True] :: [Bool]  
  
Nothing  :: Maybe a;    Just 42  :: Maybe Int  
Just 'a'  :: Maybe Char; Just ada :: Maybe Person  
  
Left 42   :: Either Int Char; Right 'a' :: Either Int Char  
Left adam :: Either Person (Person -> [Anschrift])  
Right ws  :: Either Person (Person -> [Anschrift])
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

475/170

# Summentypen (fgs.)

Beispiel:

```
data BildSchriftUndTontraeger =  
    Buch Autor Titel Verlag Auflage Lieferbar  
    | E_Buch Autor Titel Verlag LizenzBisJahr  
    | DVD Titel Hauptdarsteller Regisseur Sprachen  
    | CD Kuenstler Titel Spieldauer
```

Wertbeispiele:

```
Buch "Richard Bird" "Thinking Functionally"  
    "Cambridge University Press" 1 True  
:: BildSchriftUndTontraeger  
E_Buch "Simon Thompson" "Haskell" "Pearson" 2018  
:: BildSchriftUndTontraeger  
DVD "Der Pate" ["Marlon Brando","Al Pacino"]  
    "Francis Ford Coppola" ["Englisch","Deutsch","Italienisch"]  
:: BildSchriftUndTontraeger  
CD "Angelika Nebel" "Klaviersonaten" (1,1,48)  
:: BildSchriftUndTontraeger
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

476/170

# Summentypen (fgs.)

Beispiel:

```
data Baum = Leer
          | Wurzel Baum Int Baum
          deriving (Eq,Ord,Show)

data TBaum' = Laub Person [Anschrift]
            | Gabel' Person [Anschrift]
              TBaum' TBaum' TBaum'
```

Wertbeispiele:

```
Leer :: Baum
Wurzel Leer 42 Leer :: Baum
Wurzel (Wurzel Leer 17 Leer) 42 Leer :: Baum

adrs = [A "Sydney" "High St" 1 "Australia",
        A "Adelaide" "1st Ave" 10 "Australia"] :: [Anschrift]

t1 = Laub ada adrs :: TBaum'
t2 = Laub adam [A "Perth" "Main St" 42 "Australia"] :: TBaum'
t3 = Gabel' (P "Haskell" "Curry" Maennlich) [] t1 t2 t1 :: TBaum'
t4 = Gabel' ada adrs t2 t3 t4 :: TBaum'      -- nicht endlich!
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

# Zusammenfassung

...mit `data` ein einheitliches Sprachkonstrukt in Haskell für

- ▶ Aufzählungstypen, Produkttypen, Summentypen als Ausprägungen algebraischer Datentypen.
- ▶ oft unterschiedliche Sprachkonstrukte in anderen Sprachen, z.B. drei in Pascal (siehe Anhang D).

Aufzählungs- und Produkttypen auffassbar als

- ▶ Randfall oder Spezialfall von Summentypen.

Algebraische Datentypdeklarationen können

- ▶ rekursiv (z.B. `Baum`, `TBaum'`) und wechselseitig rekursiv (z.B. `Buerger`, `Nachbarn`) aufeinander Bezug nehmen.
- ▶ rekursive Typen ermöglichen es, Werte potentiell nicht beschränkter Größe (z.B. `t4 :: TBaum'`) zu konstruieren.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.2.5

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

478/170

# Kapitel 5.3

## Funktionen auf algebraischen Datentypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

**5.3**

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

# Funktionen auf algebraischen Datentypen

...werden üblicherweise mittels **Musterpassung** (engl. **pattern matching**) definiert.

In der Folge betrachten wir einige Beispiele.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

**5.3**

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

480/1700



# Beispiel einer Funktion

...auf `BildSchriftUndTonTraeger`-Daten, die `Selektorfunktion` `titel`:

```
titel :: BildSchriftUndTonTraeger -> Titel
titel (Buch aut tit verl aufl lieferb) = tit
titel (E_Buch aut tit verl lizenz)     = tit
titel (DVD t _ _ _) = t
titel (CD _ t _)    = t

titel (Buch "Richard Bird" "Thinking Functionally"
      "Cambridge University Press" 1 True)
->> "Thinking Functionally" :: Titel
```

Aufrufbeispiele:

```
titel (E_Buch "Simon Thompson" "Haskell" "Pearson" 2018)
->> "Haskell" :: Titel

titel (DVD "Der Pate" ["Marlon Brando","Al Pacino"]
      "Francis Ford Coppola" ["Englisch","Deutsch","Italienisch"]
->> "Der Pate" :: Titel

titel (CD "Angelika Nebel" "Klaviersonaten" (1,1,48))
->> "Klaviersonaten" :: Titel
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

481/170

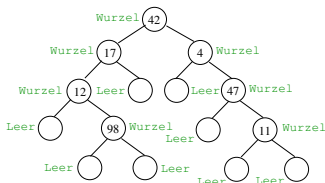
# Binärbaumwerte: Konventionelle Darstellung

## Erinnerung:

Zwei Werte vom Typ

Baum

Leer



```
data Baum
```

```
= Leer
```

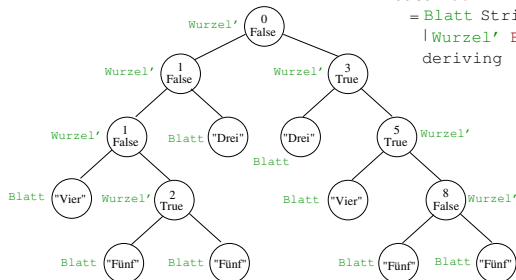
```
| Wurzel Baum Int Baum
```

```
deriving (Eq, Ord, Show)
```

Zwei Werte vom Typ

Baum'

Blatt "Eins"



```
data Baum'
```

```
= Blatt String
```

```
| Wurzel' Baum' Int Bool Baum'
```

```
deriving (Eq, Ord, Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

482/170

# Beispiele für Funktionen auf Binärbäumen

...die **Summenfkt.** `summeMarken` und die **Tiefenfkt.** `tiefe`:

```
summeMarken :: Baum -> Int
```

```
summeMarken Leer = 0
```

```
summeMarken (Wurzel ltb n rtb)
```

```
    = n + summeMarken ltb + summeMarken rtb
```

```
tiefe :: Baum' -> Int
```

```
tiefe (Blatt _) = 1
```

```
tiefe (Wurzel' ltb _ _ rtb)
```

```
    = 1 + max (tiefe ltb) (tiefe rtb)
```

## Aufrufbeispiele:

```
summeMarken Leer ->> 0
```

```
summeMarken (Wurzel Leer 2 (Wurzel Leer 3 Leer)) ->> 5
```

```
tiefe (Blatt "Fun") ->> 1
```

```
tiefe (Wurzel' (Blatt "Fun") 4 False
```

```
    (Wurzel' (Blatt "Prog") 11 True (Blatt ""))) ->> 3
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

483/170

# Beispiele von Funktionen auf

...Bürgernetzwerkdaten, die Funktionen `verwandteNachbarn` und `verwandteNachbarnNamen`:

```
type Verwandt = Bool
type Bekannt  = Bool
data Wohnform = EFH | ZFH | MFH | DH | RH | HH | PH
               deriving (Eq,Ord,Show)

data Buerger   = B Person Wohnform Nachbarn Freunde
               deriving (Eq,Ord,Show)
data Nachbarn  = N [(Buerger,Verwandt,Bekannt)]
               deriving (Eq,Ord,Show)
data Freunde   = F [Buerger]
               deriving (Eq,Ord,Show)

verwandteNachbarn :: Buerger -> [Person]
verwandteNachbarn (B _ _ (N ls) _)
  = [p | (B p _ _ _,verwandt,_) <- ls, verwandt == True]

verwandteNachbarnNamen :: Buerger -> [(Nachname,Vorname)]
verwandteNachbarnNamen (B _ _ (N ls) _)
  = [(nn,vn) | (B (P vn nn _) _ _ _,vw,_) <- ls, vw == True]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

484/170

# Kapitel 5.4

## Feldsyntax

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

**5.4**

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

# Transparente, sprechende Typdeklarationen

...drei Möglichkeiten bieten sich an, **transparente und sprechende Datentypdeklarationen** in **Haskell** zu erreichen:

- ▶ Kommentierung
- ▶ Typsynonyme
- ▶ Feldsyntax (Verbundtypsyntax)

...mit dem Zusatzvorteil

- ▶ 'geschenkter' Selektorfunktionen
- ▶ wesentlich vereinfachter weiterer Verarbeitungsfkt.

# Transparente, sprechende Typdeklarationen

...mittels **Kommentierung**:

```
newtype Gb          = Gb (String,String,String)
                      deriving (Eq,Ord,Show)

data G               = M | W deriving (Eq,Ord,Show)

data Meldedaten = Md String  -- Vorname
                          String -- Nachname
                          Gb    -- Geboren (tt,mm,jjjj)
                          G      -- Geschlecht (m/w)
                          String -- Gemeinde
                          String -- Strasse
                          Int     -- Hausnummer
                          Int     -- PLZ
                          String  -- Land
                          deriving (Eq,Ord,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

487/170

# Transparente, sprechende Typdeklarationen

...mittels Typsynonymen:

```
type Vorname      = String
type Nachname     = String
type Ziffernfolge = String
type Zf           = Ziffernfolge
newtype Gb        = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)
type Geboren      = Gb
data G            = M | W deriving (Eq,Ord,Show)
type Geschlecht   = G
type Gemeinde     = String
type Strasse      = String
type Hausnummer   = Int
type PLZ          = Int
type Land         = String

data Meldedaten   = Md Vorname Nachname Geboren
                    Geschlecht Gemeinde Strasse
                    Hausnummer PLZ Land deriving (Eq,Ord,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Kap. 10

488/170



# Transparente, sprechende Typdeklarationen

...mittels **Feldsyntax** (oder **Verbundtypsyntax**):

```
type Ziffernfolge = String
type Zf           = Ziffernfolge
data G            = M | W deriving (Eq,Ord,Show)
newtype Gb        = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)

data Meldedaten   = Md { vorname    :: String,
                        nachname    :: String,
                        geboren     :: Gb,
                        geschlecht  :: G,
                        gemeinde    :: String,
                        strasse     :: String,
                        hausnummer  :: Int,
                        plz         :: Int,
                        land        :: String
                      } deriving (Eq,Ord,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

489/170

# Transparente, sprechende Typdeklarationen

...typgleiche Felder können in der **Feldsyntax** durch Beistrich  
getrennt zusammengefasst werden:

```
type Ziffernfolge = String
type Zf           = Ziffernfolge
data G            = M | W deriving (Eq,Ord,Show)
newtype Gb        = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)

data PersDaten = PD { vorname,
                     nachname,
                     gemeinde,
                     strasse,
                     land      :: String,
                     geboren   :: Gb,
                     geschlecht :: G,
                     hausnummer,
                     plz       :: Int
                     } deriving (Eq,Ord,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

490/170

# Transparente, sprechende Typdeklarationen

...Feldnamen in Alternativen dürfen wiederholt werden, wenn ihr Typ für alle Vorkommen ident ist:

```
type Ziffernfolge = String
type Zf           = Ziffernfolge
data G            = M | W deriving (Eq,Ord,Show)
newtype Gb        = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)
data Meldedaten   = Md { vorname,
                        nachname,
                        gemeinde,
                        strasse,
                        land      :: String,
                        geboren    :: Gb,
                        geschlecht :: G,
                        hausnummer,
                        plz        :: Int
                        }
                  | KurzMd { vorname,
                           nachname :: String
                           } deriving (Eq,Ord,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

491/170

# Transparente, sprechende Typdeklarationen

...mittels **Kommentar**, **Typsynonymen** und **Feldsyntax**:

```
type Vorname      = String
type Nachname     = String
type Ziffernfolge = String
type Zf           = Ziffernfolge
newtype Gb        = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)
type Geboren      = Gb
data G            = M | W deriving (Eq,Ord,Show)
type Geschlecht   = G
type Gemeinde     = String
type Strasse      = String
type Hausnummer   = Int
type PLZ          = Int
type Land         = String
```

```
data Meldedaten = Md { vorname      :: Vorname,
                      nachname     :: Nachname,
                      geboren      :: Geboren, -- (tt,mm,jjjj)
                      geschlecht   :: Geschlecht,
                      gemeinde     :: Gemeinde,
                      strasse      :: Strasse,
                      hausnummer  :: Hausnummer,
                      plz          :: PLZ,
                      land         :: Land
                      } deriving (Eq,Ord,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

492/170

# Selektorfunktionen

...über **Kommentierung** bzw. **Typsynonyme** definierte Typen erfordern üblicherweise **musterbasiert-definierte Selektor-, Wertsetzungs- und Werterzeugungsfunktionen**:

```
gibVorname :: Meldedaten -> Vorname
gibVorname (Md vn _ _ _ _ _ _ _) = vn

gibNachname :: Meldedaten -> Nachname
gibNachname (Md _ nn _ _ _ _ _ _) = nn

...

gibPLZ :: Meldedaten -> PLZ
gibPLZ (Md _ _ _ _ _ _ plz _) = hsnr

gibLand :: Meldedaten -> Land
gibLand (Md _ _ _ _ _ _ _ land) = land
```

...für **Meldedaten** sind auf diese Weise **9 Selektorfunktionen** separat zu schreiben.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

493/170

# Wertsetzungsfunktionen

...in gleicher Weise gilt dies für Wertsetzungsfunktionen:

```
setzeVorname :: Vorname -> Meldedaten -> Meldedaten
setzeVorname vn (Md _ nn geb gs gem str hsnr plz land)
  = Md vn nn geb gs gem str hsnr plz land

setzeNachname :: Nachname -> Meldedaten -> Meldedaten
setzeNachname nn (Md vn _ geb gs gem str hsnr plz land)
  = Md vn nn geb gs gem str hsnr plz land

...

setzePLZ :: PLZ -> Meldedaten -> Meldedaten
setzePLZ plz (Md vn nn geb gs gem str hsnr _ land)
  = Md vn nn geb gs gem str hsnr plz land

setzeLand :: Land -> Meldedaten -> Meldedaten
setzeLand land (Md vn nn geb gs gem str hsnr plz _)
  = Md vn nn geb gs gem str hsnr plz land
```

...9 Wertsetzungsfunktionen für Meldedaten.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

494/170

# Werterzeugungsfunktionen

...ebenso für Werterzeugungsfunktionen:

```
undefiniert = undefiniert -- Auswertung terminiert nicht!
```

```
erzeugeMdMitVorname :: Vorname -> Meldedaten
```

```
erzeugeMdMitVorname vorname
```

```
  = Md vorname undefiniert undefiniert undefiniert  
    undefiniert undefiniert undefiniert undefiniert  
    undefiniert
```

```
...
```

```
erzeugeMdMitLand :: Land -> Meldedaten
```

```
erzeugeMdMitVorname land
```

```
  = Md undefiniert undefiniert undefiniert undefiniert  
    undefiniert undefiniert undefiniert undefiniert  
    land)
```

...9 Werterzeugungsfunktionen für Meldedaten.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

495/170

# Selektorfunktionen bei Feldnamenverwendung

...die Selektorfunktionen sind durch die Feldnamen gegeben und damit 'geschenkt':

```
gibVorname :: Meldedaten -> Vorname
gibVorname = vorname

gibNachname :: Meldedaten -> Nachname
gibNachname = nachname

...

gibPLZ :: Meldedaten -> PLZ
gibPLZ = plz

gibLand :: Meldedaten -> Land
gibLand = land
```

**Anmerkung:** Die Funktionen `gibVorname`, `gibNachname`, etc., sind *de facto* nur Synonyme bzw. Aliase der Feldnamen `vorname`, `nachname`, etc.; ihre Einführung deshalb im Grunde obsolet.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

496/170



# Wertsetzungsfkt. bei Feldnamenverwendung

...Feldnamen erlauben eine wesentlich knappere Schreibweise der Wertsetzungsfunktionen:

```
setzeVorname :: Vorname -> Meldedaten -> Meldedaten
```

```
setzeVorname vn md = md {vorname = vn}
```

```
setzeNachname :: Nachname -> Meldedaten -> Meldedaten
```

```
setzeNachname nn md = md {nachname = nn}
```

```
...
```

```
setzePLZ :: PLZ -> Meldedaten -> Meldedaten
```

```
setzePLZ p md = md {plz = p}
```

```
setzeLand :: Land -> Meldedaten -> Meldedaten
```

```
setzeLand ld md = md {land = ld}
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

497/170

# Werterzeugungsfkt. bei Feldnamenverwendung

...dies gilt vergleichbar auch für Werterzeugungsfunktionen:

```
erzeugeMdMitVorname :: Vorname -> Meldedaten
erzeugeMdMitVorname vn = Md {vorname = vn}

erzeugeMdMitNachname :: Nachname -> Meldedaten
erzeugeMdMitNachname nn = Md {nachname = nn}

...

erzeugeMdMitPLZ :: PLZ -> Meldedaten
erzeugeMdMitPLZ p = Md {plz = p}

erzeugeMdMitLand :: Land -> Meldedaten
erzeugeMdMitLand ld = Md {land = ld}
```

...nicht genannte Felder werden automatisch\*) 'undefiniert' gesetzt.

\*) Sprachimplementierungsabhängig: 'Gute' Übersetzer und Interpretierer sollten das jedenfalls tun.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

498/170

# Setzen oder initialisieren mehrerer Felder

...auch mehrere Felder können gesetzt werden:

```
setzeVorundNachname :: Vorname -> Nachname
                      -> Meldedaten -> Meldedaten

setzeVorundNachname vn nn md
  = md {vorname=vn, nachname=nn}
      -- Nicht genannte Felder behalten ihren Wert.

...

erzeugeMdMitVorundNachname :: Vorname -> Nachname
                           -> Meldedaten

erzeugeMdMitVorundNachname vn nn
  = Md {vorname=vn, nachname=nn}
      -- Nicht genannte Felder werden 'undefiniert' gesetzt.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

499/170

# Weitere Beispiele von Feldnamenverwendungen

...liefere Vor- und Nachnamen, getrennt durch ein Leerzeichen:

```
gibVollerName :: Meldedaten -> String
gibVollerName md
  = vorname md ++ " " ++ nachname md

gibVollerName' :: Meldedaten -> String
gibVollerName' (Md {vorname = vn, nachname = nn})
  = vn ++ " " ++ nn
```

Gleichwertig ohne Feldnamenverwendung:

```
gibVollerName'' :: Meldedaten -> String
gibVollerName'' (Md vn nn _ _ _ _ _ _)
  = vn ++ " " ++ nn
```

doch weniger bequem, da

- ▶ die Zahl der Unterstriche **exakt** stimmen muss.
- ▶ **änderungsaufwändig** bei Hinzu-/Wegnahme von Feldkomponenten (alle Aufrufstellen müssen angepasst werden!)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

500/1700

# Ausblick

...in **Kapitel 10** werden wir über

- ▶ monomorphe (Daten-) Typen und Funktionen

hinausgehend

- ▶ polymorphe (Daten-) Typen

und

- ▶ polymorphe und überladene Operationen und Funktionen

besprechen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

**5.4**

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

# Kapitel 5.5

## Anwendungshinweise

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

**5.5**

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 5.5.1

## Produkttypen vs. Tupeltypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

**5.5.1**

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Produkttypen vs. Tupeltypen

...am Beispiel des Typs `Person` als `Produkt`- und als `Tupeltyp`:

## Produkttyp

```
data Person = P Vorname Nachname Geschlecht
-- Produkttyp im engeren Sinn: Konstruktor P mehrstellig
data Person = P (Vorname, Nachname, Geschlecht)
-- Produkttyp im weiteren Sinn: Konstruktor P einstellig
newtype Person = P (Vorname, Nachname, Geschlecht)
-- Kein Produkttyp im strengen Sinn: newtype statt data
```

## Tupeltyp

```
type Person = (Vorname, Nachname, Geschlecht)
```

**Offensichtlicher Unterschied:** Kein Konstruktor im Tupeltyp von `Person` wie im entsprechenden Produkttyp, hier `P`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9



# Vorteile von Produkt- gegenüber Tupeltypen

...zusammengefasst in einem Wort: **Typsicherheit**.

- ▶ Werte des Produkttyps sind **typgesichert**, da sie mit dem mit dem Konstruktor **'markiert'** sind.
- ▶ 'Zufällig' passende Werte sind deshalb nicht irrtümlich, versehentlich oder absichtlich als Wert des Produkttyps manipulierbar: **Typsicherheit!** (Vgl. frühere Beispiele zu Wertpapier-, Wasserstands- und Positionsdaten).
- ▶ **Aussagekräftigere (Typ-) Fehlermeldungen**; Typsynonyme können wg. Expansion in Fehlermeldungen fehlen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

505/170

# Vorteile von Tupel- gegenüber Produkttypen

...zusammengefasst in einem Wort: **Anwendungskomfort**.

- ▶ Auf den Grundtypen (von Typsynonymen) und Tupeln vordefinierte Funktionen stehen ohne Einschränkung zur Verfügung (z.B. `fst`, `snd`, `(+)`, `(*)`, ...).
- ▶ Tupelwerte erfordern keine Konstruktoren und sind deshalb (geringfügig) kompakter (weniger Schreibaufwand für den Programm Quelltext).
- ▶ (Geringfügig) höhere Ausführungsperformanz, da 'ein-' und 'auspacken' von Tupelwerten entfällt.

**Hinweis:** Bei einstelligen Produkttypen ist statt einer `data`- auch eine `newtype`-Deklaration möglich; hier kein Effizienzverlust, da der Konstruktor nur zur Übersetzungszeit für die Überprüfung der Typkorrektheit benötigt wird.

# Beispiel: Telefonbuch

...mittels `type`-Deklarationen ausschließlich:

```
type Vorname      = String
type Nachname     = String
type Spitzname    = String
type Name         = (Vorname,Nachname)
type Telefonnr    = Int
type Telefonbuch  = [(Name,Telefonnr)]
```

```
gibTelnr :: Name -> Telefonbuch -> Telefonnr
gibTelnr name ((name',tnr):tb_rest)
  | name == name' = tnr
  | otherwise     = gibTelnr name tb_rest
gibTelnr _ []     = error "Telefonnummer unbekannt"
```

```
gibSpitzname :: Telefonnr -> Telefonbuch -> Spitzname
gibSpitzname tnr (((vn,_),tnr'):tb_rest)
  | tnr == tnr'   = vn++"ilein"
  | otherwise     = gibSpitzname tnr tb_rest
gibSpitzname _ [] = error "Spitzname unbekannt"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Beispiel: Telefonbuch

...mittels `newtype`-Deklarationen ausschließlich:

```
newtype Vorname      = Vn String deriving (Eq,Show)
newtype Nachname     = Nn String deriving (Eq,Show)
newtype Spitzname    = Sn String deriving (Eq,Show)
newtype Name         = N (Vorname,Nachname) deriving (Eq,Show)
newtype Telefonnr    = T Int  deriving (Eq,Show)
newtype Telefonbuch  = Tb [(Name,Telefonnr)] deriving (Eq,Show)
```

```
gibTelnr :: Name -> Telefonbuch -> Telefonnr
gibTelnr (N name) (Tb ((N name',T tnr):tb_rest))
  | name == name'    = T tnr
  | otherwise        = gibTelnr (N name) (Tb tb_rest)
gibTelnr _ (Tb []) = error "Telefonnummer unbekannt"
```

```
gibSpitzname :: Telefonnr -> Telefonbuch -> Spitzname
gibSpitzname (T tnr) (Tb ((N (Vn vn,_),T tnr'):tb_rest))
  | tnr == tnr'      = Sn (vn++"ilein")
  | otherwise        = gibSpitzname (T tnr) (Tb tb_rest)
gibSpitzname _ (Tb []) = error "Spitzname unbekannt"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Vergleich der type- und newtype-Varianten

## type-Deklarationen:

- ▶ Die 'eentlichen' Werte (der Typen `String`, `Int`) liegen frei zutage und können direkt in Mustern bezeichnet werden.
- ▶ Ergebnisse können unmittelbar zurückgegeben werden.

## newtype-Deklarationen:

- ▶ Typkonstruktoren (`Vn`, `Nn`, `Sn`, `N`, `T`, `Tb`) sind integraler Bestandteil von Werten und müssen deshalb explizit in Argumentmustern angegeben werden, um die 'eentlichen' Werte (der Typen `String`, `Int`) freizulegen und bezeichnen zu können.
- ▶ Bei der Rückgabe von Ergebnissen muss der 'eentliche' Wert (der Typen `String`, `Int`) in den passenden Konstruktor 'eingepackt' werden (in `gibTelnr`: `'T tnr'` statt `'tnr'`; in `gibSpitzname`: `'Sn vn++"ilein"'` statt `'vn++"ilein"'`).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 5.5.2

## Typsynonyme vs. neue Typen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

**5.5.2**

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Eigenschaften von `newtype`-Deklarationen

`newtype`-Deklarationen entsprechen im Hinblick auf

- ▶ **Typsicherheit** `data`-Deklarationen:  
Datenwerte liegen geschützt und markiert hinter `newtype`-Datenkonstruktoren.
- ▶ **Performanz** `type`-Deklarationen:  
`newtype`-Datenkonstruktoren werden ausschließlich zur Übersetzungszeit für die Typprüfung benötigt; nicht zur Laufzeit.

`newtype`-Deklarationen vereinen somit die

- ▶ **besten Eigenschaften** von `data`- und `type`-Deklarationen.

**Aber:** Typsicherheit ohne Zusatzaufwand zur Laufzeit hat einen Preis!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

511/170

# Beschränkungen von newtype-Deklarationen

`newtype`-Deklarationen sind beschränkt auf Deklarationen mit

- ▶ genau einem (Datenwert-) Konstruktor mit genau einem (Daten-) Feld ('there is no free lunch!').

Beispiele:

```
newtype Person = P {  
    1 1-stelliger Konstruktor 1 Feld  
    Vorname, Nachname, Geboren  
} -- Zulässig!
```

```
newtype Person = P {  
    1 3-stelliger Konstruktor 3 Felder  
    Vorname Nachname Geboren  
} -- Nicht zulässig!
```



# Kapitel 5.5.3

Faustregel zur Wahl von `type`, `newtype`,  
`data`

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

**5.5.3**

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Faustregel zur Wahl von type-Deklarationen

type-Deklarationen führen einen

- ▶ neuen Namen für einen existierenden Typ ein.

Sind sinnvoll, wenn

- ▶ durch 'sprechendere' Typnamen die Transparenz und Verständlichkeit von Signaturen erhöht werden soll.
- ▶ auf den Komfort, die auf dem Grundtyp definierten Funktionen weiterzubnutzen, nicht verzichtet werden soll.

Allerdings:

- ▶ Keine höhere Typsicherheit.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Faustregel z. Wahl v. `newtype`-Deklarationen

`newtype`-Deklarationen führen einen

- ▶ neuen Typ für einen existierenden Typ ein.

Sind sinnvoll, wenn

- ▶ zusätzlich zu 'sprechenderen' Typnamen auch Typsicherheit erreicht werden soll.
- ▶ 'Erhöhte' Laufzeitkosten algebraischer Typen vermieden werden sollen (und `newtype` anwendbar ist (s.u.)).
- ▶ Typen zu Instanzen von Typklassen gemacht werden sollen (siehe Kap. 4.3 und Kap. 11.4).

Allerdings:

- ▶ Eingeschränkte Anwendungsmöglichkeit. Nur möglich für Typen mit genau einem Datenwertkonstruktor und genau einem Datenfeld.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Faustregel zur Wahl von data-Deklarationen

data-Deklarationen erlauben in freier Weise

- ▶ neue Typen zu kreieren.

Sind sinnvoll (bzw. nötig), wenn

- ▶ Typsicherheit benötigt wird.
- ▶ eine newtype-Deklaration ausscheidet, weil ein neuer bislang nicht existierender Typ mit mehr als einem Konstruktor oder mehr als einem Datenfeld benötigt wird.

Allerdings:

- ▶ Leicht erhöhte Verarbeitungskosten gegenüber newtype-Deklarationen durch Konstruktorbehandlung nicht nur zur Übersetzungs-, sondern auch zur Laufzeit.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Summa summarum

**type-** vs. **newtype-** und **data-**Deklarationen:

...**type-**Deklarationen

- ▶ wo 'angemessen' und 'ausreichend', zusätzliche Typsicherheit nicht erforderlich ist.
- ▶ **newtype-** und **data-**Deklarationen, wo zusätzliche Typsicherheit nötig und unverzichtbar ist.

**newtype-** vs. **data-**Deklarationen:

...**newtype-**Deklarationen

- ▶ wo möglich; **data-**Deklarationen, wo nötig.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 5.6

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

**5.6**

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9




Teil IV

Kap. 10

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 5 (1)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 7, Eigene Typen und Typklassen definieren)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 8, Benutzerdefinierte Datentypen)
-  Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 4, Algebraische Datentypen)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 8.1, Type declarations; Kapitel 8.2, Data declarations; Kapitel 8.3, Newtype declarations; Kapitel 8.4, Recursive types)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 5 (2)

-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 7, Making our own Types and Type Classes; Kapitel 12, Monoids – Wrapping an Existing Type into a New Type)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 12, Konstruktion von Datenstrukturen)
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006. (Kapitel 6, Typen; Kapitel 8, Polymorphe und abhängige Typen; Kapitel 9, Spezifikationen und Typklassen)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9




Teil IV

Kap. 10

520/170



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 5 (3)

-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 2, Types and Functions; Kapitel 3, Defining Types, Streamlining Functions – Defining a New Data Type, Type Synonyms, Algebraic Data Types)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 14, Algebraic types)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 14, Algebraic types)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

521/170

# Kapitel 6

## Muster und mehr

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

**Kap. 6**

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Muster und mehr

## Muster, Musterpassung (Kap. 6.1)

- ▶ Elementare Datentypen
- ▶ Tupeltypen
- ▶ Listentypen
  - ▶ []-Muster
  - ▶ (p:ps)-Muster, (p:(q:qs))-Muster, etc.
  - ▶ `as`-Muster (engl. `as` pattern)
- ▶ Algebraische Datentypen

## Listenkomprehension (Kap. 6.2)

## Konstruktoren, Operatoren (Kap. 6.3)

- ▶ Begriffsbestimmung und Vergleich am Beispiel von Listen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-523/170

# Kapitel 6.1

## Muster, Musterpassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

**6.1**

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster, Musterpassung

Muster sind

- ▶ (syntaktische) Ausdrücke zur Beschreibung der **Struktur von Werten**.

Musterpassung (engl. **pattern matching**) erlaubt

- ▶ in Funktionsdefinitionen mithilfe einer Folge von Mustern Alternativen auszuwählen. Dabei werden die Muster in einer festen Reihenfolge (von oben nach unten) durchprobiert; **passt** die Struktur eines (Argument-) Werts auf ein Muster, wird diese Alternative ausgewählt.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 6.1.1

## Muster für Werte elementarer Datentypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

**6.1.1**

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster für Werte elementarer Datentypen (1)

...am Beispiel von Funktionen auf Wahrheitswerten:

Konstanten und Joker als Muster:

```
nicht :: Bool -> Bool
```

```
nicht True  = False
```

```
nicht _     = True
```

```
und :: Bool -> Bool -> Bool
```

```
und True True = True
```

```
und _ _       = False
```

```
oder :: Bool -> Bool -> Bool
```

```
oder False False = False
```

```
oder _ _         = True
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster für Werte elementarer Datentypen (2)

Konstanten, Variablen und Joker als Muster:

```
nund :: Bool -> Bool -> Bool
```

```
nund True True = False
```

```
nund _ _      = True
```

```
noder :: Bool -> Bool -> Bool
```

```
noder False False = True
```

```
noder _ _          = False
```

```
xoder :: Bool -> Bool -> Bool
```

```
xoder a b = a /= b
```

```
wennDannSonst :: Bool -> a -> a -> a
```

```
wennDannSonst True t _ = t
```

```
wennDannSonst False _ e = e
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9



# Muster für Werte elementarer Datentypen (3)

...am Beispiel von Funktionen auf **ganzen Zahlen**:

**Konstanten**, **Variablen** und **Joker** als Muster:

```
mult :: Int -> Int -> Int
```

```
mult _ 0 = 0
```

```
mult 0 _ = 0
```

```
mult m 1 = m
```

```
mult 1 n = n
```

```
mult m n = m * n
```

```
potenz :: Integer -> Integer -> Integer
```

```
potenz _ 0 = 1
```

```
potenz m n = m * potenz m (n-1)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Muster für Werte elementarer Datentypen sind:

- ▶ **Konstanten:** 0, 42, 3.14, 'c', True,...  
~> ein Wert **passt** auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- ▶ **Variablen:** b, m, n, t, e,...  
~> jeder Wert **passt** (**und** ist rechtsseitig verwendbar).
- ▶ **Joker** (eng. **wild card**): \_  
~> jeder Wert **passt** (**aber** ist rechtsseitig nicht verwendbar).

# Kapitel 6.1.2

## Muster für Werte von Tupeltypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

**6.1.2**

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster für Werte von Tupeltypen

...am Beispiel von **polymorphen** Funktionen und Funktionen auf **ganzen Zahlen**:

**Konstanten**, **Variablen** und **Joker** als Muster:

$\text{fst}' :: (a,b,c) \rightarrow a$

$\text{fst}' (\textcolor{red}{x}, \_, \_) = x$

$\text{snd}' :: (a,b,c) \rightarrow b$

$\text{snd}' (\_, \textcolor{red}{y}, \_) = y$

$\text{thd}' :: (a,b,c) \rightarrow c$

$\text{thd}' (\_, \_, \textcolor{red}{z}) = z$

$\text{binom}' :: (\text{Integer}, \text{Integer}) \rightarrow \text{Integer}$

$\text{binom}' (\textcolor{red}{n}, \textcolor{red}{k})$

|  $k==0 \mid n==k = 1$

| otherwise =  $\text{binom}' (n-1, k-1) + \text{binom}' (n-1, k)$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

532/1700

# Zusammenfassung

Muster für Werte von Tupeltypen sind:

- ▶ **Konstanten:**  $(0,0)$ ,  $(0, \text{"Null"})$ ,  $(3.14, \text{"pi"}, \text{True})$ , ...  
~> ein Wert **passt** auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- ▶ **Variablen:**  $t$ ,  $t1$ , ...  
~> jeder Wert **passt** (**und** ist rechtsseitig verwendbar).
- ▶ **Joker** (eng. **wild card**):  $\_$   
~> jeder Wert **passt** (**aber** ist rechtsseitig nicht verwendbar).
- ▶ **Kombinationen aus Konstanten, Variablen, Jokern:**  
 $(m, n)$ ,  $(\text{True}, n, \_)$ ,  $(\_, (m, \_, n), 3.14, k, \_)$ , ...  
~> ein Wert **passt**, wenn er strukturell mit dem Muster übereinstimmt.

# Kapitel 6.1.3

## Muster für Werte von Listentypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

**6.1.3**

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster für Werte von Listentypen (1)

Konstanten als Muster; Konstruktormuster mit Konstanten,  
Variablen und Jokern:

```
sum :: [Int] -> Int
sum []      = 0
sum (0:xs)  = sum xs
sum (x:xs)  = x + sum xs
```

```
mult :: [Int] -> Int
mult []      = 1
mult (0:_)   = 0
mult (1:xs)  = mult xs
mult (x:xs)  = x * mult xs
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster für Werte von Listentypen (2)

Konstanten, Variablen und Joker als Muster; Konstruktormuster mit Jokern:

```
kopf :: [a] -> a
```

```
kopf (x:_) = x
```

```
rest :: [a] -> [a]
```

```
rest (_,xs) = xs
```

```
leer :: [a] -> Bool
```

```
leer [] = True
```

```
leer _ = False
```

```
verbinde :: [a] -> [a] -> [a] -> [a]
```

```
verbinde ps qs rs = ps ++ qs ++ rs
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9



# Muster für Werte von Listentypen (3)

Konstanten und Joker als Muster; Konstruktormuster mit Variablen und Jokern:

```
nimm :: Int -> [a] -> [a]    -- entspricht vordef.  
nimm m ys = case (m,ys) of    -- Fkt. take  
    (0,_)      -> []  
    (_,[])     -> []  
    (n,(x:xs)) -> x : nimm (n - 1) xs
```

```
streiche :: Int -> [a] -> [a] -- entspricht vordef.  
streiche m ys = case (m,ys) of -- Fkt. drop  
    (0,_)      -> ys  
    (_,[])     -> []  
    (n,(_:xs)) -> streiche (n - 1) xs
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster für Werte von Listentypen (4)

**Konstruktor**muster erlauben auch, 'endlich tief' in eine Liste hineinzusehen:

```
maxElem :: Ord a => [a] -> a
maxElem []           = error "Ungueltige Eingabe"
maxElem (y:[])       = y
maxElem (x:y:ys)     = maxElem ((max x y) : ys)
```

```
length' :: [a] -> Int
length' [] = 0
length' (u:v:w:x:y:z:zs) = 6 + length' zs
length' (v:w:x:y:z:zs) = 5 + length' zs
length' (w:x:y:z:zs) = 4 + length' zs
length' (x:y:z:zs) = 3 + length' zs
length' (y:z:zs) = 2 + length' zs
length' _ = 1
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster für Werte von Listentypen (5)

Konstanten, Variablen und Joker als Muster für Zeichenreihenwerte; Konstruktormuster mit Variablen für Zeichenreihen:

```
anfuegen :: String -> String -> String
anfuegen "" t = t
anfuegen s "" = s
anfuegen s t  = s ++ t
```

```
istPrefix :: String -> String -> Bool
istPrefix "" _ = True
istPrefix (c:_) "" = False
istPrefix (c:cs) (d:ds) = (c==d) && istPrefix cs ds
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Zusammenfassung (1)

**Muster** für Werte von **Listentypen**, speziell **Zeichenreihen**, sind:

- ▶ **Konstanten**: `[]`, `""`, `[1,2,3]`, `[1..50]`, `['a'..'z']`, `[True,False,True,True]`, `"aeiou"`, ...  
~> ein Wert **passt** auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- ▶ **Variablen**: `p`, `q`, `ps`, `qs`...  
~> jeder Wert **passt** (**und** ist rechtsseitig verwendbar).
- ▶ **Joker** (eng. **wild card**): `_`  
~> jeder Wert **passt** (**aber** ist rechtsseitig nicht verwendbar).

# Zusammenfassung (2)

## ► Konstruktormuster:

$(\langle \text{muster\_listenkopf} \rangle : \langle \text{muster\_listenrest} \rangle),$   
 $(p:ps), (p:q:qs), \dots$

$\rightsquigarrow$  ein Listenwert  $ls$  passt auf das Konstruktormuster  $(\langle \text{muster\_listenkopf} \rangle : \langle \text{muster\_listenrest} \rangle)$ , wenn  $\langle \text{muster\_listenkopf} \rangle$  und  $\langle \text{muster\_listenrest} \rangle$  gültige Musterausdrücke für Listenköpfe und Listenreste sind,  $ls$  nicht leer ist, der Kopf von  $ls$  strukturell mit  $\langle \text{muster\_listenkopf} \rangle$  übereinstimmt und der Rest von  $ls$  mit  $\langle \text{muster\_listenrest} \rangle$ .

$ls$  passt strukturell auf das Konstruktormuster  $(p:ps)$  bzw.  $(p:q:qs)$  mit  $p, ps, q, qs$  Variablenmuster, wenn  $ls$  nicht leer ist bzw. mindestens 2 Elemente enthält.

# Kapitel 6.1.4

## Muster für Werte algebraischer Datentypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

**6.1.4**

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster für Werte algebraischer Datentypen (1)

Konstanten entsprechend 0-stelliger Konstruktoren als Muster:

```
type Zeichenreihe = [Char]
data Jahreszeiten = Fruehling | Sommer
                  | Herbst | Winter

wetter :: Jahreszeiten -> Zeichenreihe
wetter Fruehling = "Launisch"
wetter Sommer  = "Sonnig"
wetter Herbst   = "Windig"
wetter Winter   = "Frostig"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster für Werte algebraischer Datentypen (2)

## Konstruktormuster mit Variablen:

```
type Zett = Int
data Ausdruck = Opd Zett
               | Add Ausdruck Ausdruck
               | Sub Ausdruck Ausdruck
               | Quad Ausdruck

eval :: Ausdruck -> Zett
eval (Opd n)      = n
eval (Add e1 e2)  = (eval e1) + (eval e2)
eval (Sub e1 e2)  = (eval e1) - (eval e2)
eval (Quad e)     = (eval e)^2
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9



# Muster für Werte algebraischer Datentypen (3)

Konstanten als Muster; Konstruktormuster mit Variablen und Jokern:

```
type Zett = Int
data Baum a b = Blatt a
               | Wurzel b (Baum a b) (Baum a b)

tiefe :: (Baum a b) -> Zett
tiefe (Blatt _)      = 1
tiefe (Wurzel _ l r) = 1 + max (tiefe l) (tiefe r)

data Liste a = Leer | Kopf a (Liste a)

listenLaenge :: Liste a -> Zett
listenLaenge Leer      = 0
listenLaenge (Kopf _ xs) = 1 + listenLaenge xs
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Muster für Werte algebraischer Datentypen (4)

**Konstruktormuster** erlauben wie für Listen auch in Werte algebraischer Datentypen 'endlich tief' hineinzusehen:

```
type Zett = Int
data Baum = Blatt Zett
          | Gabel Zett Baum Baum

putzig :: Baum -> Zett
putzig (Blatt 7) = 42
putzig (Blatt n) = n*n
putzig (Gabel n (Blatt m) (Gabel p (Blatt q) (Blatt r)))
      = n+m+p+q+r
putzig (Gabel n (Gabel _ (Blatt q) (Blatt r)) (Blatt _))
      = n*(q+r)
putzig _ = 0
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Zusammenfassung

Muster für Werte algebraischer Typen sind:

- ▶ **Konstanten:** Sommer, Winter, Empty,...  
↪ ein Wert passt auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- ▶ **Variablen:** e, e1, e2, t,...  
↪ jeder Wert passt (und ist rechtsseitig verwendbar).
- ▶ **Joker** (eng. wild card): \_  
↪ jeder Wert passt (aber ist rechtsseitig nicht verwendbar).
- ▶ **Konstruktormuster:** (Opd e), (Add e1 e2), (Blatt' 7), (Blatt' n), (Blatt' \_), (Gabel 42 1 r), (Kopf \_ hs),...  
↪ ein Wert passt strukturell auf das Konstruktormuster, wenn seine Struktur mit der des Musters übereinstimmt.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

547/170

# Kapitel 6.1.5

## Das als-Muster

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

**6.1.5**

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Das als-Muster (1)

Oft sehr nützlich ist das sog. **als-Muster** (engl. **as pattern**).

Betrachte folgendes Beispiel:

```
nichtleerePostfixe :: String -> [String]
nichtleerePostfixe (c:cs)
    = (c:cs) : nichtleerePostfixe cs
nichtleerePostfixe _ = []

nichtleerePostfixe "Curry"
    ->> ["Curry", "urry", "rry", "ry", "y"]
```

Die rechte Seite der ersten definierenden Gleichung nimmt Bezug auf

- ▶ das gesamte strukturierte Argument: **(c:cs)**
- ▶ einen Teil des strukturierten Arguments: **cs**

# Das als-Muster (2)

Das **als-Muster** erlaubt dies einfacher auszudrücken:

```
nichtleerePostfixe :: String -> [String]
nichtleerePostfixe s@(_:cs)
                    = s : nichtleerePostfixe cs
nichtleerePostfixe _ = []
```

Das **als-Muster** `s@(_:cs)` (`@` gelesen als '**als**' (engl. '**as**')) bietet je einen **Namen** an für

- ▶ das gesamte Argument, in diesem Beispiel: `s`
- ▶ für die relevanten strukturellen Komponenten des Arguments, in diesem Beispiel für den Rest der Liste, wenn die Argumentliste nicht leer ist: `cs`
- ▶ Auch möglich: **Alle Strukturkomponenten** namentlich zu bezeichnen, z.B. `s@(c:cs)`

# Vorteile aus der Verwendung des als-Musters

...anhand des Beispiels der Funktion `nichtleerePostfixe`:

- ▶ Mittels `s` lässt sich auf das gesamte Argument Bezug nehmen; mittels `cs` auf die strukturelle Komponente des Listenrests, wenn die Argumentliste nicht leer ist.
- ▶ Die Verwendung des `als-Musters` führt deshalb wie in diesem Beispiel meist zu einfacheren und übersichtlicheren Definitionen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Zum Vergleich

...beide Definitionen noch einmal gegenübergestellt:

- ▶ Mit als-Muster:

```
nichtleerePostfixe :: String -> [String]
nichtleerePostfixe s@(_:cs)
  = s : nichtleerePostfixe cs
nichtleerePostfixe _ = []
```

- ▶ Ohne als-Muster:

```
nichtleerePostfixe :: String -> [String]
nichtleerePostfixe (c:cs)
  = (c:cs) : nichtleerePostfixe cs
nichtleerePostfixe _ = []
```



# Weitere Beispiele (1)

...Listen und als-Muster.

Die Funktion `listTransform` mit `als`-Muster:

```
listTransform :: [a] -> [a]
listTransform ys@(x:xs) = (x : ys) ++ xs
```

Zum Vergleich `listTransform` ohne `als`-Muster:

```
listTransform :: [a] -> [a]
listTransform (x:xs) = (x : (x : xs)) ++ xs
```

## Weitere Beispiele (2)

...Tupel und als-Muster.

Die Funktion `tausche` mit `als`-Muster:

```
tausche :: Eq a => (a,a) -> (a,a)
tausche paar@(x,y)
  | x /= y      = (y,x)
  | otherwise   = paar
```

Zum Vergleich `tausche` ohne `als`-Muster:

```
tausche :: Eq a => (a,a) -> (a,a)
tausche (x,y)
  | x /= y      = (y,x)
  | otherwise   = (x,y)
```

# Weitere Beispiele (3)

...Tupel und als-Muster.

Die Funktion `tauscheBedingt` mit `als`-Muster:

```
tauscheBedingt :: (a,Bool,a) -> (a,Bool,a)
tauscheBedingt tripel@(x,b,y)
  | b      = (y,b,x)
  | not b  = tripel
```

Zum Vergleich `tauscheBedingt` ohne `als`-Muster:

```
tauscheBedingt :: (a,Bool,a) -> (a,Bool,a)
tauscheBedingt (x,b,y)
  | b      = (y,b,x)
  | not b  = (x,b,y)
```

# Generell

...ist das **als**-Muster

- ▶ über **Listen** und **Tupel** hinaus

allgemein für

- ▶ **algebraische Datentypen** mit **strukturierten Werten**

nützlich.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

**6.1.5**

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 6.1.6

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

**6.1.6**

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Vorteile musterbasierter Funktionsdefinitionen

## Musterbasierte Funktionsdefinitionen

- ▶ sind elegant.
- ▶ führen (i.a.) zu knappen, gut lesbaren Spezifikationen.

Zur Illustration: Die Funktion `binom'` mit Mustern sowie ohne Muster mittels Standardselektoren:

```
binom' :: (Integer,Integer) -> Integer
binom' (n,k)      -- mit Mustern
  | k==0 || n==k = 1
  | otherwise    = binom' (n-1,k-1) + binom' (n-1,k)

binom' :: (Integer,Integer) -> Integer
binom' p          -- ohne Muster mit Std.-Selektoren
  | snd(p)==0 || snd(p)==fst(p) = 1
  | otherwise = binom' (fst(p)-1,snd(p)-1)
                + binom' (fst(p)-1,snd(p))
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Allerdings

...musterbasierte Funktionsdefinitionen können

- ▶ zu subtilen Fehlern führen.
- ▶ Programmänderungen/-weiterentwicklungen erschweren, 'bis hin zur Tortur', etwa beim Hinzukommen eines oder mehrerer weiterer Parameter.

(siehe dazu: Peter Pepper. [Funktionale Programmierung in OPAL, ML, Haskell und Gofer](#). Springer-Verlag, 2. Auflage, 2003, S. 164.)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.1.6

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

# Kapitel 6.2

## Listenkomprehension

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

**6.2**

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-560/170



# Listenkomprehension (1)

...ein charakteristisches, elegantes und ausdruckskräftiges Sprachmittel

► funktionaler Programmiersprachen

das die Mengenbildungsoperation aus der Mathematik auf Listen nachbildet und ohne Parallele in Sprachen anderer Paradigmen ist.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

- 561/170

# Listenkomprehension (2)

...erlaubt **Listen** auf eine Weise zu beschreiben, in der ihre Elemente durch

► **filtern**, **testen**, **transformieren** der Elemente **anderer Listen** erzeugt werden.

Zur **Illustration**: Eine Reihe von Beispielen zur Verwendung von **Listenkomprehension** in

- **Ausdrücken**
- **Funktionsdefinitionen**
- **Zeichenreihen** (als speziellen Listen)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

562/170

# Listenkompensation in Ausdrücken (1)

Zwei Listen:

```
ns1 = [1,2,3,4]
```

```
ns2 = [1,2,4,7,8,11,12,42]
```

Ein Generator, eine Transformation:

```
[ 3 * n | n <- ns1 ]
```

```
->> [3,6,9,12]
```

```
[ square n | n <- ns2 ]
```

```
->> [1,4,16,49,64,121,144,1764]
```

```
[ isPrime n | n <- ns2 ]
```

```
->> [False,True,False,True,False,True,False,False]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-563/170

# Listenkomprehension in Ausdrücken (2)

Ein Generator, ein bzw. zwei Tests, eine Transformation:

```
[fac n | n <- ns2, isPowOfTwo n]  
->> [1,2,24,40320]
```

```
[id n | n <- ns2, isPowOfTwo n, n >= 5]      -- ", "  
->> [8]                                       -- steht für "und"
```

Zwei Generatoren, ein Filter, zwei Tests, eine Transformation:

```
[((m,n),m+n) | m <- ns1, n <- tail ns2,  
               m <= 2, n <= 7]  
->> [((1,2),3),((1,4),5),((1,7),8),  
     ((2,2),4),((2,4),6),((2,7),9)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-564/170

# Listenkompensation in Ausdrücken (3)

Zwei Generatoren, zwei Filter, ein Test, eine Transformation:

```
[fib ((+) m n) | m <- take 3 ns1, n <- drop 5 ns2,  
                (odd (m+n) || (m*n)>100) ]  
->> [fib ((+) m n) | m <- [1,2,3], n <- [11,12,42],  
                    (odd (m+n) || (m*n)>100) ]  
->> [fib (1+12),fib (1+42),  
      fib (2+11),  
      fib (3+12),fib (3+42)]  
->> [fib 13, fib 43, fib 13, fib 15, fib 45]  
->> ...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-565/170

# Listenkomprehension in Fkt.-Definitionen (1)

Abstandsberechnung vom Ursprung einer Liste von Punkten:

```
type Point = (Float,Float)
distanceFromOrigin :: [Point] -> [Float]
distanceFromOrigin ps
    = [sqrt (squ x + squ y) | (x,y) <- ps]
distanceFromOrigin [(3.0,4.0),(1.0,1.0),(-1.0,3.0)]
->> [5.0,1.414,3.1623]
```

Test auf Ungradheit und Gradheit aller Listenelemente:

```
allOdd :: [Integer] -> Bool
allOdd ns = ([n | n <- ns, isOdd n] == ns)
allOdd [2..22] ->> False
allEven :: [Integer] -> Bool
allEven ns = ([n | n <- ns, isOdd n] == [])
allEven [2,4..22] ->> True
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

566/170

# Listenkomprehension in Fkt.-Definitionen (2)

```
grabCapVowels :: String -> String
grabCapVowels cs = [c | c <- cs, isCapVowel c]

isCapVowel :: Char -> Bool
isCapVowel 'A' = True
isCapVowel 'E' = True
isCapVowel 'I' = True
isCapVowel 'O' = True
isCapVowel 'U' = True
isCapVowel _  = False

grabCapVowels "Alles Eint Informatik Ohne Unterschied!"
->> "AEIOU"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

567/170

# Listenkomprehension in Fkt.-Definitionen (3)

## QuickSort:

```
quickSort :: [Integer] -> [Integer]
quickSort [] = []
quickSort (n:ns) = quickSort [m | m <- ns, m <= n]
                  ++ [n]
                  ++ quickSort [m | m <- ns, m > n]
```

**Anmerkung:** Funktionsanwendung bindet stärker als Listenkonstruktion; deshalb Klammerung des Musters `(n:ns)` in der zweiten definierenden Gleichung `quickSort (n:ns) = ...`

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-568/170



# Listenkompensation und Zeichenreihen

...Zeichenreihen sind in Haskell ein Typsynonym für Listen von Zeichen:

```
type String = [Char]
```

Beispiel:

```
"Haskell" == ['H','a','s','k','e','l','l']
```

Für Zeichenreihen als spezielle Listen stehen deshalb dieselben

- ▶ Funktionen
- ▶ Komprehensionsmechanismen

zur Verfügung wie für allgemeine Listen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-569/170

# Listenfunktionen für Zeichenreihen

## Beispiele für Funktionen auf Zeichenreihen als Listen:

```
"Haskell"!!3 ->> 'k'
```

```
(!!) "Haskell" 3 ->> 'k'
```

```
take 5 "Haskell" ->> "Haske"
```

```
drop 5 "Haskell" ->> "ll"
```

```
length "Haskell" ->> 7
```

```
zip "Haskell" [1,2,3] ->> [('H',1),('a',2),('s',3)]
```

```
"Haskell" 'zip' [1,2,3] ->> [('H',1),('a',2),('s',3)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-570/170

# Listenkompensation für Zeichenreihen

## Zählen der Kleinbuchstaben in einer Zeichenreihe:

```
lowers :: String -> Int
lowers cs = length [c | c <- cs, isLower c]

lowers "Haskell" ->> 6
```

## Zählen der Vorkommen eines bestimmten Zeichens in einer Zeichenreihe:

```
count :: Char -> String -> Int
count c cs = length [c | d <- cs, d == c]

count 's' "Mississippi" ->> 4
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-571/170

# Kapitel 6.3

## Konstruktoren, Operatoren

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

**6.3**

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-572/170

# Konstruktoren vs. Operatoren

Konstruktoren führen zu **eindeutigen** Darstellungen von Werten, **Operatoren** nicht.

Beispiel:

- ▶ **(:)** ist (einziger) **Konstruktor** für Listen.
- ▶ **(++)** ist (einer von vielen) **Operator(en)** auf Listen.

Betrachte:

```
[42,17,4] == (42:(17:(4:[])))  -- Eindeutige Dar-
                                -- stellung von [42,17,4]
                                -- mittels des Konstruktors (:).
```

```
[42,17,4] == [42,17] ++ [] ++ [4]  -- Viele Darstel-
                                == [42] ++ [17,4] ++ []  -- lungen von
                                == [42] ++ [] ++ [17,4]  -- [42,17,4]
                                == ...  -- mittels des
                                -- Operators (++) .
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-573/170

# Bemerkungen

- ▶ Die Darstellung `(42:(17:(4:[])))` deutet an, dass eine Liste **ein** Objekt ist; erzwungen durch die Typstruktur.
- ▶ Anders in **imperativen/objektorientierten Sprachen**: Listen sind dort nur **indirekt existent**, nämlich bei 'geeigneter' Verbindung von Elementen durch Zeiger.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-574/170

# Muster, Konstruktoren und Operatoren

...Operatoren (wie z.B. `(++)` für Listen) implizieren keine **Zerlegungseindeutigkeit** von Werten und dürfen deshalb

- ▶ nicht in Mustern

verwendet werden. In Mustern dürfen ausschließlich

- ▶ Konstruktoren

verwendet werden, die diese Zerlegungseindeutigkeit (über Listen hinaus generell für algebraische Datentypen) garantieren.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-575/170

# Veranschaulichung

```
deleteTwo :: (Char,Char) -> String -> String
deleteTwo _ ""          = ""
deleteTwo _ (s:[])      = [s]
deleteTwo (c,d) (s:(t:ts))
  | [c,d] == [s,t]      = deleteTwo (c,d) ts
  | otherwise            = s : deleteTwo (c,d) (t:ts)
```

...ist **sinnvoll** und **zulässig**, weil das Muster **(s:(t:ts))** die Struktur 'passender' Argumentwerte und damit das Resultat eindeutig festlegt.

```
deleteTwo :: (Char,Char) -> String -> String
deleteTwo _ ""          = ""
deleteTwo _ (s:[])      = [s]
deleteTwo (c,d) s@([s1]++[s2])
  | [c,d] == [s1]        = deleteTwo (c,d) s2
  | otherwise             = head s : deleteTwo (c,d) (tail s)
```

...ist **nicht sinnvoll** und **unzulässig**, weil das 'Muster **([s1]++[s2])**' die Wahl von **[s1]** und **[s2]** zur Zerlegung von **s** nicht eindeutig festlegt und sich je nach Zerlegung ein anderes Resultat ergäbe.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-576/1700



# Kapitel 6.4

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

**6.4**

Teil III

Kap. 7

Kap. 8





Kap. 9

Teil IV

Kap. 10

Kap. 11

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 6 (1)

-  Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2. Auflage, 1998.  
(Kapitel 4.2, List operations)
-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 5.1.4, Automatische Erzeugung von Listen)
-  Antonie J. T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 7.4, List comprehensions)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 4.4, Pattern matching; Kapitel 5, List comprehensions)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9





Teil IV

Kap. 10

Kap. 11

-578/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 6 (2)

-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 3, Syntax in Functions – Pattern Matching)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 12, Barcode Recognition – List Comprehensions)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 13, Mehr syntaktischer Zucker)
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 2.4, Lists; Kapitel 4.1, Lists)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-579/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 6 (3)



Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999.  
(Kapitel 5.4, Lists in Haskell; Kapitel 5.5, List comprehensions; Kapitel 7.1, Pattern matching revisited; Kapitel 7.2, Lists and list patterns; Kapitel 9.1, Patterns of computation over lists; Kapitel 17.3, List comprehensions revisited)



Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.  
(Kapitel 5.5, Lists in Haskell; Kapitel 5.6, List comprehensions; Kapitel 7.1, Pattern matching revisited; Kapitel 7.2, Lists and list patterns; Kapitel 10.1, Patterns of computation over lists; Kapitel 17.3, List comprehensions revisited)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

-580/170

# Teil III

## Applikative Programmierung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

**Teil III**

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Applikatives Programmieren

...im strengen Sinn:

- ▶ **Applikatives Programmieren** ist ein Programmieren auf dem Niveau von elementaren Werten.
- ▶ Mit Konstanten, Variablen und Funktionsapplikationen werden Ausdrücke gebildet, deren Werte stets elementar sind.
- ▶ Durch explizite Abstraktion nach gewissen Variablen erhält man Funktionen.

Damit:

- ▶ Tragendes Konzept **applikativer Programmierung** zur Programmerstellung ist die **Funktionsapplikation**, d.h. die Anwendung von Funktionen auf (elementare) Werte.

Wolfram-Manfred Lippe. **Funktionale und Applikative Programmierung**. eXamen.press, 2009, Kapitel 1.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Funktionales Programmieren

...im **strengen Sinn**:

- ▶ **Funktionales Programmieren** ist ein Programmieren auf Funktionsniveau.
- ▶ Ausgehend von Funktionen werden mit Hilfe von Funktionen höherer Ordnung neue Funktionen gebildet.
- ▶ Es treten im Programm keine Applikationen von Funktionen auf elementare Werte auf.

Damit:

- ▶ Tragendes Konzept **funktionaler Programmierung** zur Programmerstellung ist die **Bildung neuer Funktionen** aus gegebenen Funktionen **mit Hilfe von Funktionen höherer Ordnung**.

Wolfram-Manfred Lippe. **Funktionale und Applikative Programmierung**. eXamen.press, 2009, Kapitel 1.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe III.1

Kommen Sie zwischendurch, spätestens am Ende von [Kapitel 10.6](#), wieder auf diese Charakterisierung und Abgrenzung [applikativer](#) und [funktionaler Programmierung](#) zurück (s.a. [Übungsaufgabe 10.6.1](#)).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

**Teil III**

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Kapitel 7

## Rekursion

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

**Kap. 7**

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Kapitel 7.1

## Motivation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

**7.1**

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

586/170

# Rekursion

...zentrales Mittel funktionaler Sprachen

- ▶ Wiederholungen auszudrücken (Beachte: Wir haben keine Anweisungen und deshalb auch keine Schleifen in funktionalen Sprachen).

Rekursives Vorgehen

- ▶ führt oft auf sehr elegante Lösungen, die konzeptuell wesentlich einfacher und intuitiver sind als schleifenbasierte imperative Lösungen (Typische Beispiele: Quicksort, Türme von Hanoi).

Rekursion insgesamt so wichtig, dass eine

- ▶ Klassifizierung von Rekursionstypen zweckmäßig ist.

...eine solche Klassifizierung nehmen wir in der Folge vor.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

587/170

# Quicksort, Türme von Hanoi

...zwei Beispiele, für die rekursives Vorgehen auf besonders

- ▶ intuitive, einfache und elegante Lösungen

führt:

- ▶ Quicksort: Schnelles Sortieren.
- ▶ Türme von Hanoi: Umschichten eines Turms aus 50 goldenen Scheiben nach bestimmten Regeln, womit nach einer hindischen Sage eine Gruppe von Mönchen seit dem Anbeginn der Zeit betraut ist.\*

\* Die Sage berichtet, dass das Ende der Welt gekommen ist, wenn die Mönche ihre Aufgabe vollendet haben.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

588/170

# Kapitel 7.1.1

## Schnelles Sortieren, Quicksort

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

**7.1.1**

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

# Quicksort

...bereits **besprochen** (vgl. Kap. 1.2.1):

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
quickSort (n:ns) =
  -- Elem. aus ns kleiner n nach links & rekursiv sortiert:
  quickSort smaller
  -- n selbst steht automatisch sortiert in der 'Mitte':
  ++ [n]
  -- Elem. aus ns grösser n nach rechts & rekursiv sort.:
  ++ quickSort larger
  where smaller = [m | m <- ns, m <= n]
        larger  = [m | m <- ns, m > n]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

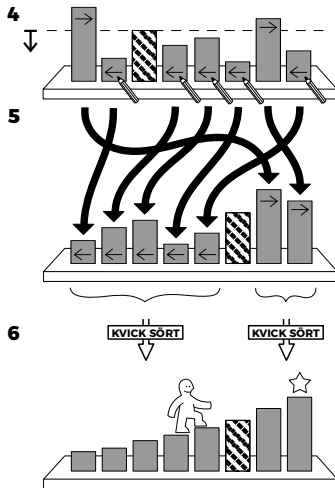
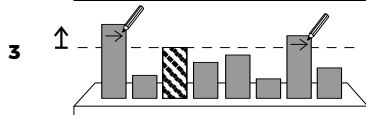
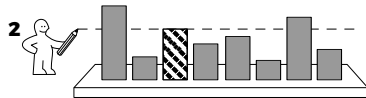
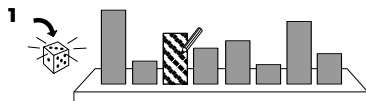
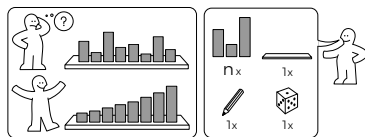
590/1700

# Veranschaulichung im IKDEA-Stil

## KVICK SÖRT

idea-instructions.com/quick-sort/  
v1.1, CC by-nc-sa 4.0

IDEA



# Kapitel 7.1.1

## Türme von Hanoi

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

**7.1.2**

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV



# Türme von Hanoi

...Aufgabe und Regelwerk:

Ausgangssituation:

Gegeben sind drei Stapelplätze A, B und C. Auf Platz A liegt ein Stapel paarweise verschieden großer goldener Scheiben, die mit von unten nach oben abnehmender Größe übereinander geschichtet sind.

Aufgabe:

Schichte den Scheibenstapel von Platz A auf Platz C um unter Ausnutzung von Platz B als Zwischenablage.

Regelwerk:

Bei jedem Umschichtungszug darf stets nur eine Scheibe bewegt werden; nie darf eine größere Scheibe oberhalb einer kleineren Scheibe auf einem der drei Plätze zu liegen kommen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

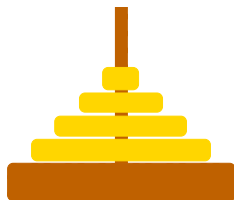
Kap. 9

Teil IV

593/170

# Veranschaulichung: Türme von Hanoi (1)

Ausgangssituation:



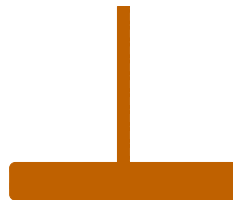
Stapelplatz A

Ausgangsstapel



Stapelplatz B

Zwischenablage



Stapelplatz C

Zielstapel

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

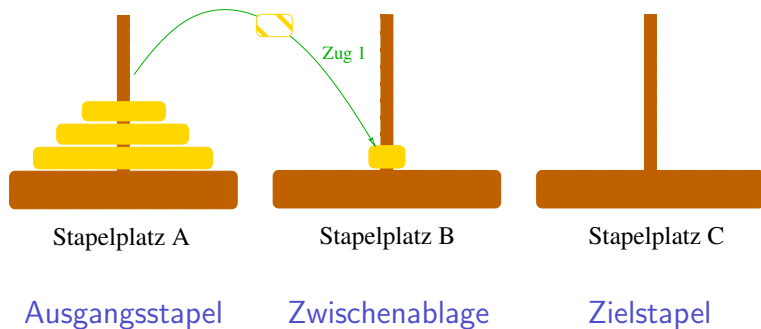
Kap. 9

Teil IV

594/170

# Veranschaulichung: Türme von Hanoi (2)

Nach einem Zug:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

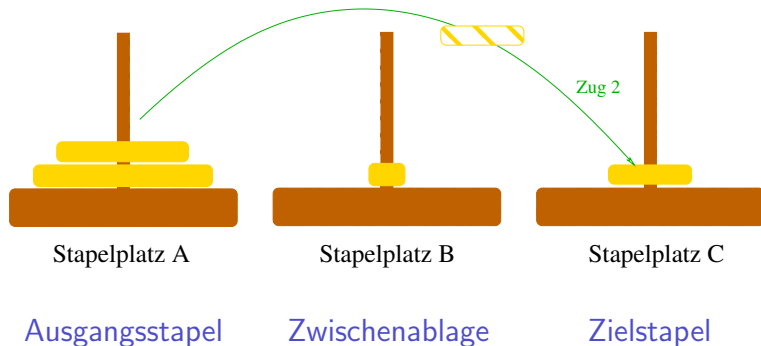
Kap. 9

Teil IV

595/170

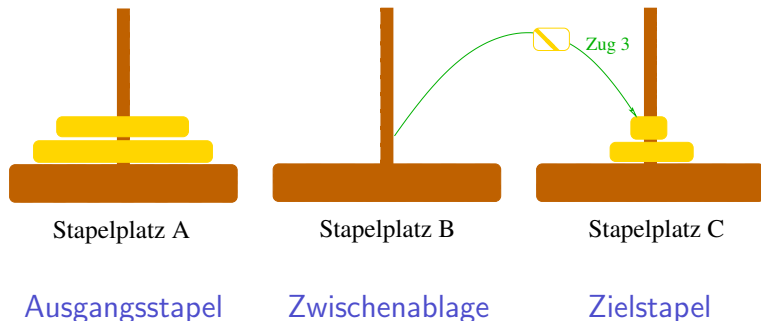
# Veranschaulichung: Türme von Hanoi (3)

Nach zwei Zügen:



# Veranschaulichung: Türme von Hanoi (4)

Nach drei Zügen:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

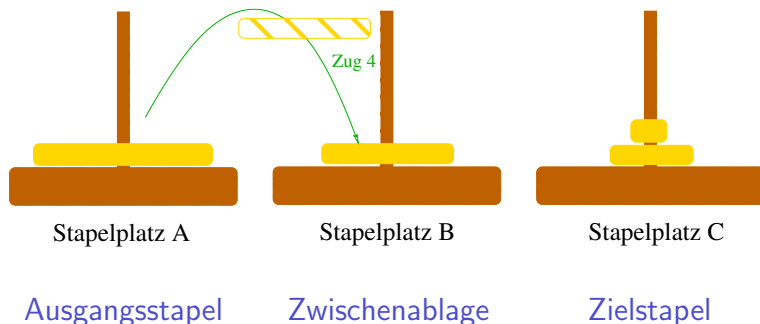
Kap. 9

Teil IV

597/170

# Veranschaulichung: Türme von Hanoi (5)

Nach vier Zügen:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

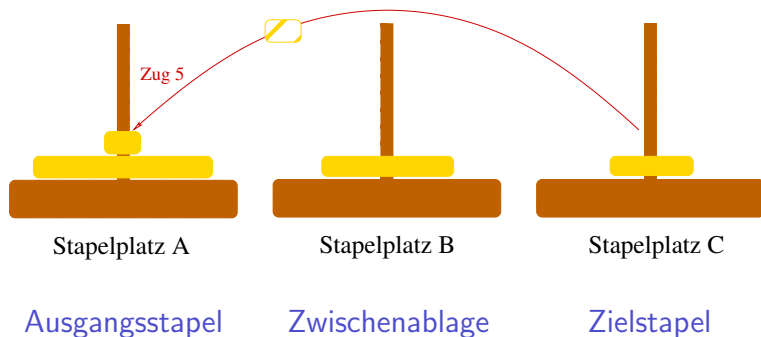
Kap. 9

Teil IV

598/170

# Veranschaulichung: Türme von Hanoi (6)

Nach fünf Zügen:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

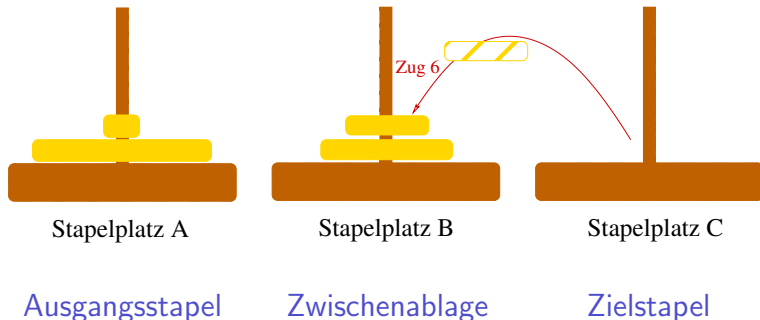
Kap. 9

Teil IV

599/170

# Veranschaulichung: Türme von Hanoi (7)

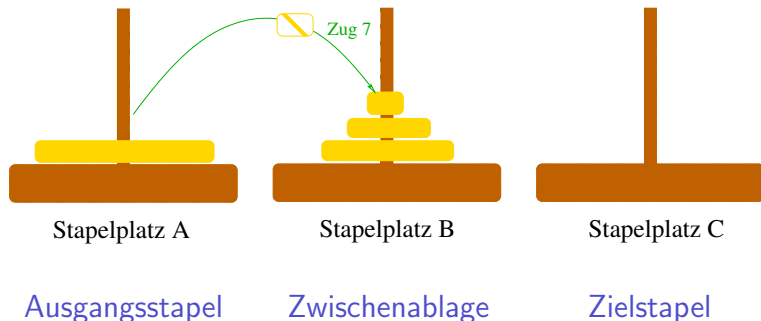
Nach sechs Zügen:





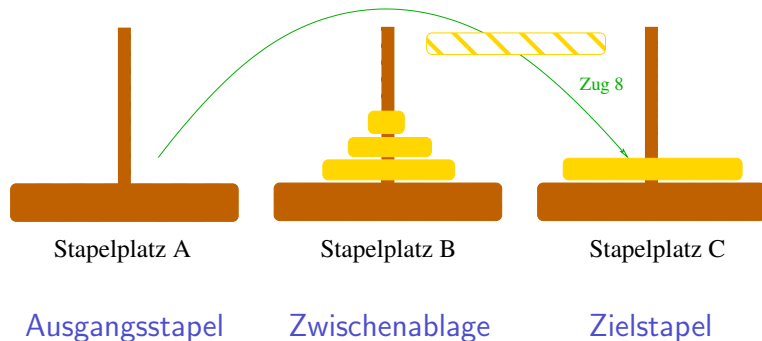
# Veranschaulichung: Türme von Hanoi (8)

Nach sieben Zügen:



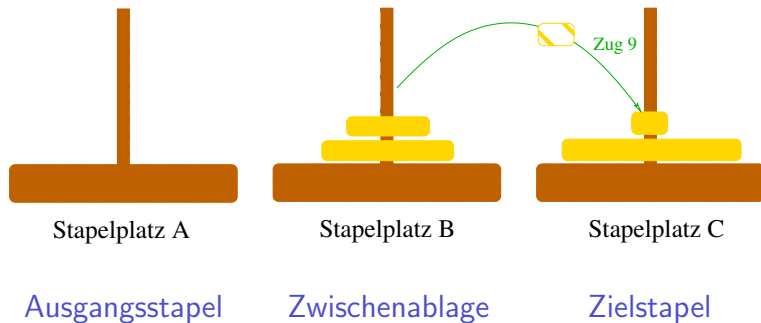
# Veranschaulichung: Türme von Hanoi (9)

Nach acht Zügen:



# Veranschaulichung: Türme von Hanoi (10)

Nach neun Zügen:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

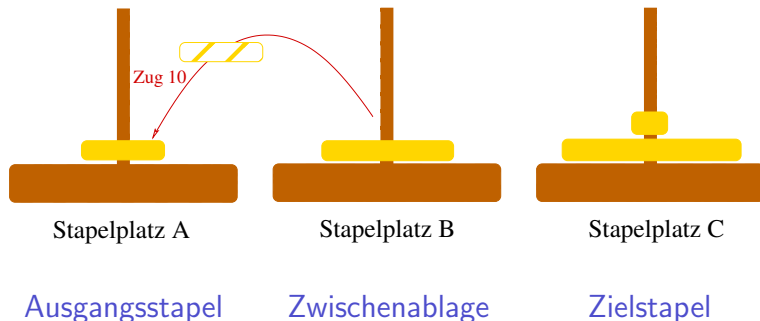
Kap. 9

Teil IV

603/170

# Veranschaulichung: Türme von Hanoi (11)

Nach zehn Zügen:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

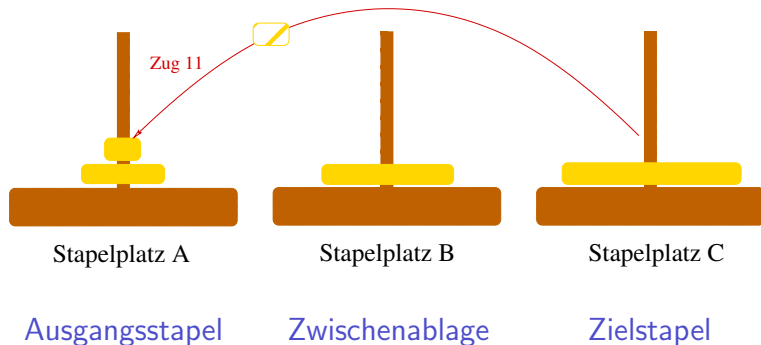
Kap. 9

Teil IV

604/170

# Veranschaulichung: Türme von Hanoi (12)

Nach elf Zügen:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

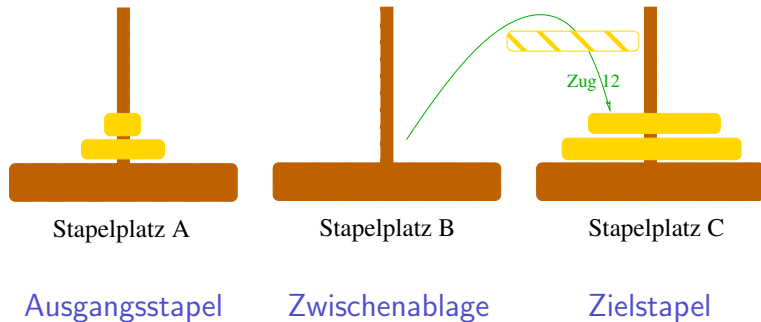
Kap. 9

Teil IV

605/170

# Veranschaulichung: Türme von Hanoi (13)

Nach zwölf Zügen:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

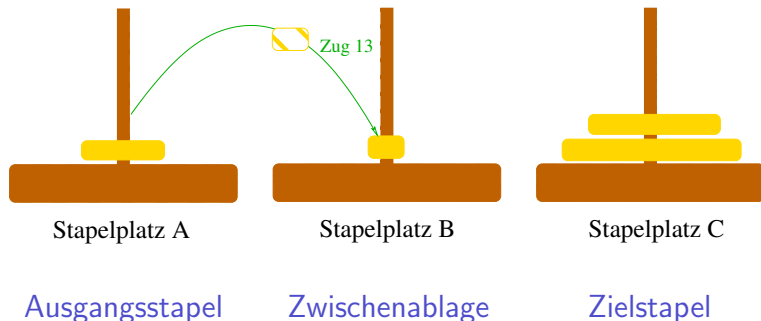
Kap. 9

Teil IV

606/170

# Veranschaulichung: Türme von Hanoi (14)

Nach dreizehn Zügen:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

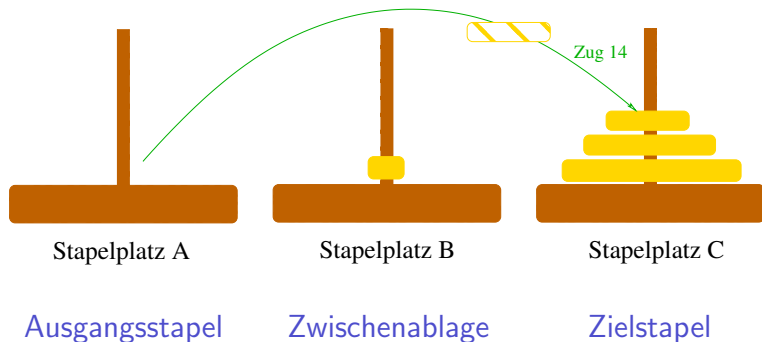
Kap. 9

Teil IV

607/170

# Veranschaulichung: Türme von Hanoi (15)

Nach vierzehn Zügen:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

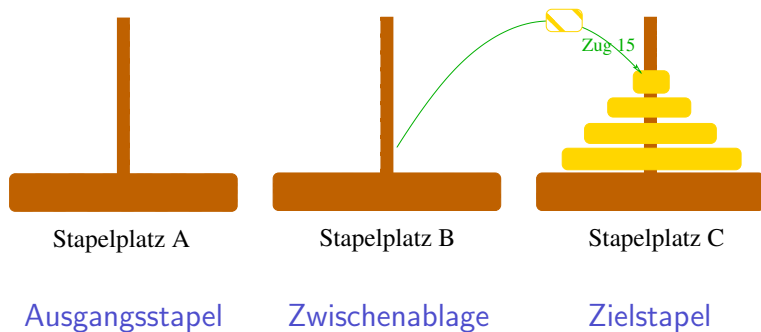
Teil IV

608/170



# Veranschaulichung: Türme von Hanoi (16)

Nach fünfzehn Zügen:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

609/170

# Veranschaulichung der Rekursionsidee (1)

**Aufgabe:** Schichte Turm  $[1, 2, \dots, N]$  von Ausgangsstapel A auf Zielstapel C um unter Verwendung von Stapelplatz B als Zwischenablage.



Stapelplatz A

Stapelplatz B

Stapelplatz C

Ausgangsstapel

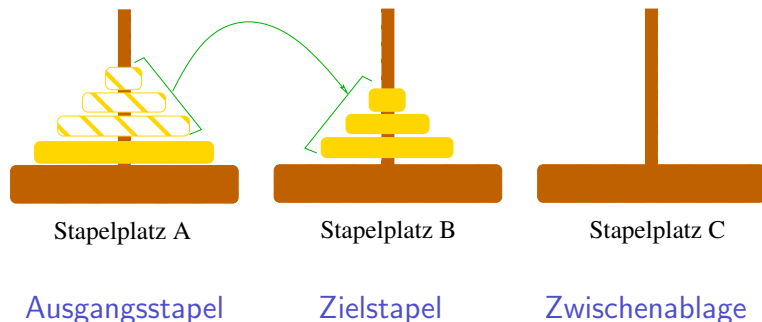
Zwischenablage

Zielstapel

# Veranschaulichung der Rekursionsidee (2)

**Schritt 1** – Scheibe  $N$  freispielen:

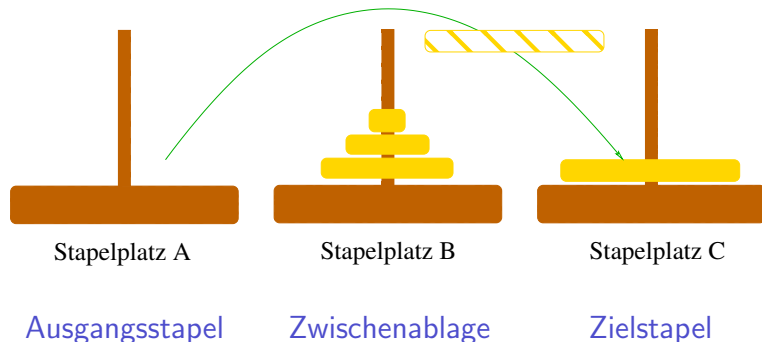
Schichte Turm  $[1, 2, \dots, N-1]$  (rekursiv) von Ausgangsstapel **A** auf Zwischenablage **B** als temporärer Zielstapel unter Ausnutzung von Stapelplatz **C** als temporäre Zwischenablage um.



# Veranschaulichung der Rekursionsidee (3)

Schritt 2 – Scheibe  $N$  spielen:

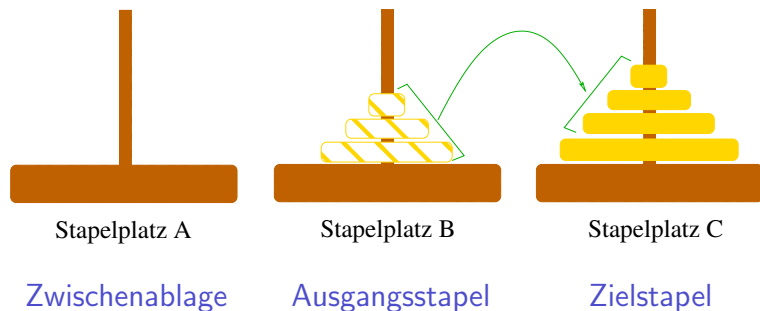
Verschiebe Scheibe  $N$  von Ausgangsstapel A auf Zielstapel C.



# Veranschaulichung der Rekursionsidee (4)

## Schritt 3 – Verbleibenden Turm umschichten:

Schichte Turm  $[1, 2, \dots, N - 1]$  (rekursiv) von Stapel B als temporärer Ausgangsstapel auf Zielstapel C unter Ausnutzung von Stapelplatz A als temporärer Zwischenablage um.



Beachte: A, B und C tauschen für Schritt 3 gegenüber Schritt 1 ihre Rollen als Ausgangs-, Ziel- und Hilfsstapelplatz.

# Zusammenfassung der Rekursionsidee

...für das Problem der Türme von Hanoi:

Der  $N$ -scheibige Turm  $[1, 2, \dots, N - 1, N]$  mit in zunehmender Größe von 1 bis  $N$  benannten Scheiben wird von Ausgangsstapel  $A$  auf Zielstapel  $C$  unter Zuhilfenahme von Stapelplatz  $B$  als Zwischenablage wie folgt umgeschichtet:

- 1) Schichte den  $(N - 1)$ -scheibigen Turm  $[1, 2, \dots, N - 1]$  rekursiv von Platz  $A$  auf Platz  $B$  unter Zuhilfenahme von Platz  $C$  als Zwischenablage um.
- 2) Verschiebe (die dadurch jetzt frei liegende unterste und größte) Scheibe  $N$  von Platz  $A$  auf Platz  $C$ .
- 3) Schichte den  $(N - 1)$ -scheibigen Turm  $[1, 2, \dots, N - 1]$  rekursiv von Platz  $B$  auf Platz  $C$  unter Zuhilfenahme von Platz  $A$  als Zwischenablage um.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

# Türme von Hanoi: Modellierung in Haskell

## Modellierung:

```
type Turmhoehe = Int      -- Anzahl Scheiben
type Scheibe   = Int      -- Scheibenidentifikator

data Stapel    = A | B | C deriving Show

type A_Stapel  = Stapel   -- Ausgangsstapel A
type Z_Stapel  = Stapel   -- Zielstapel Z
type H_Stapel  = Stapel   -- Hilfsstapel H
type Von       = Stapel   -- Stapelbezeichner
type Nach      = Stapel   -- Stapelbezeichner

hanoi :: Turmhoehe -> A_Stapel -> Z_Stapel -> H_Stapel
      -> [(Scheibe, Von, Nach)]

hanoi n a z h = <Code, der einen n-scheibigen Turm
                vom Platz a auf den Platz z um-
                schichtet unter Ausnutzung von
                Hilfplatz h als Zwischenablage>
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

615/170

# Türme von Hanoi: Implementierung in Haskell

Implementierung:

```
hanoi :: Turmhoehe -> A_Stapel -> Z_Stapel -> H_Stapel  
      -> [(Scheibe,Von,Nach)]
```

```
hanoi n a z h  
  | n == 0      = [] -- Fertig, Turm ist umgeschichtet!  
  | otherwise =  
    {- Schritt 1: Verschiebe (n-1)-Turm rek. von  
      Platz a auf Platz h über z als Hilfsplatz -}  
    (hanoi (n-1) a h z)  
    {- Schritt 2: Verschiebe Scheibe n von Platz a  
      auf Platz z -}  
  ++ [(n,a,z)]  
    {- Schritt 3: Verschiebe (n-1)-Turm rek. von  
      Platz h nach Platz z über a als Hilfsplatz -}  
  ++ (hanoi (n-1) h z a)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

616/170



# Türme von Hanoi: Aufrufe der Funktion hanoi

Main>hanoi 1 A C B

[(1,A,C)] (1 Zug für Schritt 2)

Main>hanoi 2 A C B

[(1,A,B), (1 Zug für Schritt 1)  
(2,A,C), (1 Zug für Schritt 2)  
(1,B,C)] (1 Zug für Schritt 3)

Main>hanoi 3 A C B

[(1,A,C), (2,A,B), (1,C,B), (3 Züge für Schritt 1)  
(3,A,C), (1 Zug für Schritt 2)  
(1,B,A), (2,B,C), (1,A,C)] (3 Züge für Schritt 1)

Main>hanoi 4 A C B

[(1,A,B), (2,A,C), (1,B,C), (7 Züge für Schritt 1)  
(3,A,B), (1,C,A), (2,C,B), (1,A,B),  
(4,A,C), (1 Zug für Schritt 2)  
(1,B,C), (2,B,A), (1,C,A), (7 Züge für Schritt 3)  
(3,B,C), (1,A,B), (2,A,C), (1,B,C)]

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.1.1

7.1.2

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

# Kapitel 7.2

## Rekursionstypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

**7.2**

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

# Klassifikation der Rekursionstypen

Eine Rechenvorschrift heißt **rekursiv**, wenn sie

- ▶ in ihrem Rumpf (**direkt** oder **indirekt**) aufgerufen wird.

Wir unterscheiden zwischen **Rekursion** auf

- ▶ **mikroskopischer Ebene**  
...betrachtet **einzelne Rechenvorschriften** und die syntaktische Gestalt der rekursiven Aufrufe.
- ▶ **makroskopischer Ebene**  
...betrachtet **Systeme von Rechenvorschriften** und ihre wechselseitigen Aufrufe.

# Kapitel 7.2.1

## Mikroskopische Ebene

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

**7.2.1**

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

# Rekursion auf mikroskopischer Ebene (1)

...folgende Unterscheidungen und Sprechweisen sind üblich:

## 1. Repetitive (schlichte, endständige) Rekursion

↪ pro Zweig **höchstens ein** rekursiver Aufruf und diesen stets als äußerste Operation.

Beispiel:

```
ggt :: Integer -> Integer -> Integer
```

```
ggt m n
```

```
| n == 0 = m           -- Zweig 1
| m >= n = ggt (m-n) n -- Zweig 2
| m < n  = ggt (n-m) m -- Zweig 3
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

621/170

# Rekursion auf mikroskopischer Ebene (2)

## 2. Lineare Rekursion

↪ pro Zweig **höchstens ein** rekursiver Aufruf, davon **mindestens einer** nicht als äußerste Operation.

Beispiel:

```
powerThree :: Integer -> Integer
powerThree n
  | n == 0 = 1           -- Zweig 1
  | n > 0  = 3 * powerThree (n-1)  -- Zweig 2
```

**Beachte:** In Zweig 2,  $n > 0$ , ist “**\***” die äußerste Operation, nicht **powerThree**!

# Rekursion auf mikroskopischer Ebene (3)

## 3. Baumartige (kaskadenartige) Rekursion

↪ pro Zweig können **mehrere** rekursive Aufrufe nebeneinander vorkommen.

Beispiel:

```
binom :: Integer -> Integer -> Integer
binom n k
  | k == 0 || n == k = 1                -- Zweig 1
  | otherwise =
    binom (n-1) (k-1) + binom (n-1) k -- Zweig 2
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

623/1700

# Rekursion auf mikroskopischer Ebene (4)

## 4. Geschachtelte Rekursion

↪ rekursive Aufrufe enthalten **rekursive Aufrufe** als Argumente.

Beispiel:

```
fun91 :: Integer -> Integer
fun91 n
  | n > 100 = n - 10
  | n <= 100 = fun91 (fun91 (n+11))
```

**Übungsaufgabe:** Warum heißt die Funktion wohl **fun91**?



# Zusammenfassung

...auf **mikroskopischer Ebene** sprechen wir von

- ▶ **direkter (oder unmittelbarer) Rekursion**

und unterscheiden genauer:

- ▶ **Repetitive (oder schlichte oder endständige) Rekursion**
- ▶ **Lineare Rekursion**
- ▶ **Baumartige (oder kaskadenartige) Rekursion**
- ▶ **Geschachtelte Rekursion**

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

625/170

# Übungsaufgabe 7.2.1.1

Von welchem Rekursionstyp sind die Implementierungen

- ▶ `quickSort` (Kap. 7.1.1)
- ▶ `hanoi` (Kap. 7.1.2)
- ▶ `binom`, `binom'`, `binom''` (Kap. 1.1.1)
- ▶ `zip` (Kap. 1.1.1)
- ▶ `ggt` (Kap. 1.1.1)
- ▶ `mod` (Kap. 1.1.1)

aus Kapitel 7.1 bzw. Kapitel 1.1.1?

# Kapitel 7.2.2

## Makroskopische Ebene

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

**7.2.2**

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

# Rekursion auf makroskopischer Ebene

...folgende Sprechweise ist üblich:

- Indirekte (verschränkte, wechselseitig) Rekursion  
↔ zwei oder mehr Funktionen rufen sich wechselseitig auf.

Beispiel:

```
isOdd :: Integer -> Bool
```

```
isOdd n
```

```
  | n == 0 = False
```

```
  | n > 0  = isEven (n-1)
```

```
isEven :: Integer -> Bool
```

```
isEven n
```

```
  | n == 0 = True
```

```
  | n > 0  = isOdd (n-1)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

628/1700

# Kapitel 7.2.3

## Eleganz und Effizienz, Effizienzfallen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

**7.2.3**

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

# Eleganz, Effizienz, Effizienzfallen

Viele Probleme sind **rekursiv** besonders

- ▶ **elegant lösbar** (z.B. Quicksort, Türme von Hanoi)
- ▶ jedoch **nicht immer unmittelbar effizient** ( $\neq$  effektiv!) (z.B. die naive Berechnung der Fibonacci-Zahlen)
  - ▶ **Gefahr:** (Unnötige) Mehrfachberechnungen
  - ▶ **Besonders anfällig:** Baum-/kaskadenartige Rekursion

Aus **Implementierungssicht** ist (s.a. **Anhang E**)

- ▶ **repetitive** Rekursion am **(kosten-) günstigsten**.
- ▶ **geschachtelte** Rekursion am **ungünstigsten**.

# Die Folge der Fibonacci-Zahlen

...die unendliche Folge der Fibonacci-Zahlen  $(f_i)_{i \in \mathbb{N}_0}$  ist folgendermaßen definiert:

$$f_0 = 0, f_1 = 1 \quad \text{und} \quad f_n = f_{n-2} + f_{n-1} \quad \text{für alle } n \geq 2$$

Anfang der Folge der Fibonacci-Zahlen:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

# Naive Implementierung der Fibonacci-Fkt. (1)

Die naheliegende, unmittelbar an die Definition angelehnte naive **Implementierung** mit **baumartiger Rekursion** zur Berechnung der **Fibonacci-Zahlen**:

```
fib :: Integer -> Integer
fib n
  | n == 0      = 0
  | n == 1      = 1
  | otherwise   = fib (n-2) + fib (n-1)
```

...ist **sehr, seehr laaangsaaaaaaaam** (ausprobieren!)



# Naive Implementierung der Fibonacci-Fkt. (2)

Veranschaulichung der durch die Mehrfachberechnung von Werten verursachten Ineffizienz durch manuelle Auswertung:

```
fib 0 ->> 0                                -- 1 Aufrufe von fib

fib 1 ->> 1                                -- 1 Aufrufe von fib

fib 2 ->> fib 0 + fib 1
      ->> 0 + 1
      ->> 1                                -- 3 Aufrufe von fib

fib 3 ->> fib 1 + fib 2
      ->> 1 + (fib 1 + fib 0)
      ->> 1 + (1 + 0)
      ->> 2                                -- 5 Aufrufe von fib
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

633/170

# Naive Implementierung der Fibonacci-Fkt. (3)

```
fib 4 ->> fib 2 + fib 3
      ->> (fib 0 + fib 1) + (fib 1 + fib 2)
      ->> (0 + 1) + (1 + (fib 0 + fib 1))
      ->> (0 + 1) + (1 + (0 + 1))
      ->> 3                                -- 9 Aufrufe von fib
```

```
fib 5 ->> fib 3 + fib 4
      ->> (fib 1 + fib 2) + (fib 2 + fib 3)
      ->> (1 + (fib 0 + fib 1)) +
          ((fib 0 + fib 1) + (fib 1 + fib 2))
      ->> (1 + (0 + 1)) +
          ((0 + 1) + (1 + (fib 0 + fib 1)))
      ->> (1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))
      ->> 5                                -- 15 Aufrufe von fib
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

634/170

# Naive Implementierung der Fibonacci-Fkt. (4)

```
fib 8 ->> fib 6 + fib 7
->> + (fib 4 + fib 5) + (fib 5 + fib 6)
->> ((fib 2 + fib 3) + (fib 3 + fib 4))
      + ((fib 3 + fib 4) + (fib 4 + fib 5))
->> (((fib 0 + fib 1) + (fib 1 + fib 2))
      + (fib 1 + fib 2) + (fib 2 + fib 3)))
      + (((fib 1 + fib 2) + (fib 2 + fib 3))
      + ((fib 2 + fib 3) + (fib 3 + fib 4)))
->> ...
->> 21                                -- 60 Aufrufe von fib
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

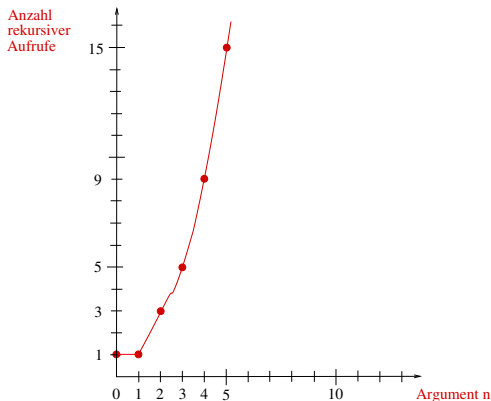
Teil IV

635/170

# Naive Implementierung der Fibonacci-Fkt. (5)

...die naive baumartig-rekursive Berechnung der Fibonacci-Zahlen führt zu äußerst vielen Mehrfachberechnungen.

Der Berechnungsaufwand wächst dabei exponentiell!



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

636/170

# Lsg. 1: Effiziente Berechnung der Fibonacci-Z.

Fibonacci-Zahlen lassen sich auf viele Arten effizient berechnen, z.B. durch:

► Rechnen auf Parameterposition!

```
fib :: Integer -> Integer
```

```
fib n = fib' n 0 1
```

```
where
```

```
fib' :: Integer -> Integer -> Integer -> Integer
```

```
fib' 0 a b = a
```

```
fib' n a b = fib' (n-1) b (a+b)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

637/170

# Lsg. 2: Effiziente Berechnung der Fibonacci-Z.

...die i.w. gleiche Idee verteilt auf mehrere Funktionen realisiert das System von Rechenvorschriften aus:

► `fibSchritt`, `fibPaar` und `fib`.

```
fibSchritt :: (Integer,Integer) -> (Integer,Integer)
```

```
fibSchritt (m,n) = (n,m+n)
```

```
fibPaar :: Integer -> (Integer,Integer)
```

```
fibPaar n
```

```
  | n == 0      = (0,1)
```

```
  | otherwise = fibSchritt (fibPaar (n-1))
```

```
fib :: Integer -> Integer
```

```
fib n = fst (fibPaar n)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

638/1700

## Lsg. 3: Effiziente Berechnung der Fibonacci-Z.

...sog. **Memo-Funktionen** führen ebenfalls zu einer effizienten Implementierung, eine Idee, die zurückgeht auf:

- Donald Michie. 'Memo' Functions and Machine Learning. Nature 218:19-22, 1968.

```
memo_fib :: [Int]
memo_fib = [fib n | n <- [0..]]      -- Memo-Liste

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = memo_fib !! (n-2) + memo_fib !! (n-1)
```

**Hinweis:** Die Listenelementzugriffsfunktion **(!!)** hat den Typ **(!!) :: [a] -> Int -> a**; deshalb ist **fib** hier über **Int** definiert, nicht über **Integer**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

639/1700

# Übungsaufgabe 7.2.3.1

Werte die effizienten Funktionen `fib` aus

- ▶ Lsg. 1
- ▶ Lsg. 2
- ▶ Lsg. 3

`händisch` für jeweils einige Werte aus, um zu sehen, wie diese Implementierungen die Berechnung der **Fibonacci-Zahlen** vornehmen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

**7.2.3**

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

640/1700



# Abhilfe bei ungünstigem Rekursionsverhalten

...oft ist folgende **Verbesserung** möglich:

- Ersetzung ungünstiger durch günstigere Rekursionsmuster!

Z.B. Rückführung **linearer** auf **repetitive Rekursion** (s.a. **Anhang E**).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

**7.2.3**

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

641/170

# Rückführung linearer auf repetitive Rek. (1)

...am Beispiel der **Fakultätsfunktion**:

Naheliegende Implementierung mittels **linearer Rekursion**:

```
fac :: Integer -> Integer
fac n
  | n == 0      = 1
  | otherwise = n * fac (n-1)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

**7.2.3**

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

642/1700

# Rückführung linearer auf repetitive Rek. (2)

Günstigere Formulierung mittels **repetitiver Rekursion** durch

► Rechnen auf Parameterposition.

```
fac :: Integer -> Integer
```

```
fac n = fac_repetitiv n 1
```

```
fac_repetitiv :: Integer -> Integer -> Integer
```

```
fac_repetitiv n resultat
```

```
  | n == 0      = resultat
```

```
  | otherwise = fac_repetitiv (n - 1) (n * resultat)
```

**Beachte:** Überlagerungen mit anderen Effekten sind möglich, so dass sich möglicherweise kein Effizienzgewinn realisiert!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.2.1

7.2.2

7.2.3

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

643/170

# Weitere Möglichkeiten zur Verbesserung

...bieten spezielle **Programmiertechniken** wie

- ▶ **Dynamische Programmierung**
- ▶ **Memoization**

**Zentrale Idee:**

- ▶ **Speicherung und Wiederverwendung** statt Neuberechnung bereits berechneter (Teil-) Ergebnisse.

(Siehe etwa die effiziente Berechnung der **Fibonacci**-Zahlen mithilfe einer **Memo**-Liste)

**Hinweis:** Dynamische Programmierung und Memoization werden in der **LVA 185.A05 Fortgeschrittene funktionale Programmierung** ausführlich behandelt.

# Kapitel 7.3

## Aufrufgraphen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

**7.3**

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Struktur von Programmen

Programme funktionaler Programmiersprachen (auch Haskell-Programme) sind i.a.

- ▶ Systeme (wechselweiser) rekursiver Rechengvorschriften, die sich hierarchisch oder/und wechselseig aufeinander abstützen.

Aufrufgraphen erleichtern es, sich über die

- ▶ Struktur von Systemen von Rechengvorschriften

Klarheit zu verschaffen.

# Aufrufgraphen

...sei  $S$  ein System von Rechenvorschriften.

## Definition 7.3.1 (Aufrufgraph)

Der **Aufrufgraph** von  $S$  ist ein Graph, der

- ▶ für jede in  $S$  deklarierte Rechenvorschrift einen **Knoten** mit dem Namen der Rechenvorschrift als Beschriftung enthält,
- ▶ eine **gerichtete Kante** vom Knoten  $f$  zum Knoten  $g$  genau dann enthält, wenn im Rumpf der zu  $f$  gehörigen Rechenvorschrift die zu  $g$  gehörige Rechenvorschrift aufgerufen wird.

# Beispiel: Aufrufgraphen (1)

...der Rechenvorschriften bzw. Systeme von Rechenvorschriften  
`add`, `add'`, `fac`, `fib`, `max` und `mx`:

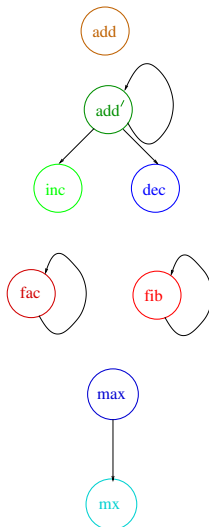
```
add :: Int -> Int -> Int
add m n = (+) m n

add' :: Int -> Int -> Int
add' m n | n == 0    = m
         | n > 0    = add' (inc m) (dec n)
         | otherwise = add' (dec m) (inc n)
where inc :: Int -> Int
      inc n = n+1
      dec :: Int -> Int
      dec n = n-1

fac :: Integer -> Integer
fac n | n == 0    = 1
      | otherwise = n * fac (n-1)

fib :: Integer -> Integer
fib n | n == 0    = 0
      | n == 1    = 1
      | otherwise = fib (n-1) + fib (n-2)

max :: Int -> Int -> Int -> Int
max p q r
  | (mx p q == p) && (p 'mx' r == p) = p
  | (mx p q == q) && (q 'mx' r == q) = q
  | otherwise                        = r
where mx :: Int -> Int -> Int -> Int
      mx p q | p >= q    = p
              | otherwise = q
```



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
648/170

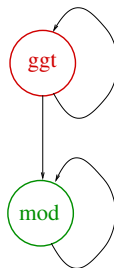


# Beispiel: Aufrufgraphen (2)

...des Systems hierarchischer Rechenvorschriften der Funktionen `ggt` und `mod`:

```
ggt :: Int -> Int -> Int
ggt m n
  | n == 0 = m
  | n > 0  = ggt n (mod m n)
```

```
mod :: Int -> Int -> Int
mod m n
  | m < n  = m
  | m >= n = mod (m-n) n
```



# Beispiel: Aufrufgraphen (3)

...des Systems wechselweise rekursiver Rechenvorschriften der Funktionen `isOdd` und `isEven`:

```
isOdd :: Integer -> Bool
```

```
isOdd n
```

```
| n == 0 = False
```

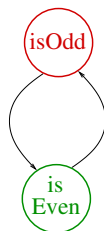
```
| n > 0  = isEven (n-1)
```

```
isEven :: Integer -> Bool
```

```
isEven n
```

```
| n == 0 = True
```

```
| n > 0  = isOdd (n-1)
```



# Beispiel: Aufrufgraphen (4)

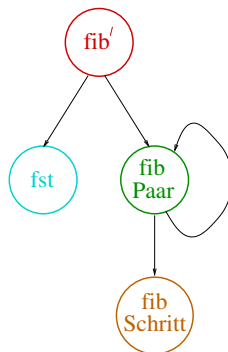
...des Systems hierarchischer Rechenvorschriften der Funktionen `fib`, `fibPaar`, `fibSchritt` und `fst`:

```
fibSchritt :: (Integer,Integer) -> (Integer,Integer)
fibSchritt (m,n) = (n,m+n)

fibPaar :: Integer -> (Integer,Integer)
fibPaar n =
  | n == 0    = (0,1)
  | otherwise = fibSchritt (fibPaar (n-1))

fib' :: Integer -> Integer
fib' n = fst (fibPaar n)

fst :: (a,b) -> a
fst (x,y) = x
```



# Interpretation von Aufrufgraphen (1)

Aus den Aufrufgraphen eines Systems von Rechenvorschriften ist u.a. ablesbar:

- ▶ **Direkte Rekursivität** einer Funktion: 'Selbstkringel'.  
(z.B. bei den Aufrufgraphen der Funktionen **fac** und **fib**)
- ▶ **Wechselweise Rekursivität** zweier Funktionen: Kreise mit zwei Kanten.  
(z.B. bei den Aufrufgraphen der Funktionen **isOdd** und **isEven**)
- ▶ **Direkte hierarchische Abstützung** einer Funktion auf eine andere: Es gibt eine Kante von Knoten **f** zu Knoten **g**, aber nicht umgekehrt.  
(z.B. bei den Aufrufgraphen der Funktionen **max** und **mx**)

# Interpretation von Aufrufgraphen (2)

- ▶ **Indirekte hierarchische Abstützung** einer Funktion auf eine andere: Knoten  $g$  ist von Knoten  $f$  über eine Folge von Kanten erreichbar, aber nicht umgekehrt.  
(z.B. bei den Aufrufgraphen der Funktionen `fib'`, `fib-Paar` und `fibSchritt`)
- ▶ **Indirekte wechselseitige Abstützung**: Knoten  $g$  ist von Knoten  $f$  über eine Folge von Kanten erreichbar und umgekehrt (Kreise mit mehr als zwei Kanten).
- ▶ **Unabhängigkeit/Isolation** einer Funktion: Knoten  $f$  hat (ggf. mit Ausnahme eines Selbstkringels) weder ein- noch ausgehende Kanten.  
(z.B. bei den Aufrufgraphen der Funktionen `add`, `fac` und `fib`)
- ▶ ...

# Übungsaufgabe 7.3.1

Gib ein **System von Rechenvorschriften** an, in dem sich zwei Funktionen **f** und **g** indirekt wechselseitig aufeinander abstützen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

**7.3**

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Kapitel 7.4

## Komplexität, Komplexitätsklassen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

**7.4**

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Rechenaufwand von Algorithmen

...als Anzahl elementarer Operationen zu (angenommenen) Einheitskosten einer imaginären Maschine.

Sei

- ▶  $\mathcal{A} : \mathcal{P} \rightarrow \mathcal{L}$  ein Algorithmus, der Probleminstanzen  $p$  aus einer Menge  $\mathcal{P}$  auf Lösungsinstanzen  $l$  aus einer Menge  $\mathcal{L}$  abbildet.
- ▶  $f_{\mathcal{A}} : \mathcal{P} \rightarrow \mathbb{N}_0$  eine Funktion, die für jede Probleminstanz  $p \in \mathcal{P}$  die Zahl elementarer Operationen angibt, die  $\mathcal{A}$  zur Lösung von  $p$  benötigt.
- ▶  $\gamma : \mathcal{P} \rightarrow \mathbb{N}_0$  eine Funktion, die jeder Probleminstanz  $p \in \mathcal{P}$  eine natürliche Zahl als Größe zuordnet.

Beispiel: Sortieren

- ▶  $\mathcal{P}$ : Menge aller Listen über ganzen Zahlen.
- ▶  $\mathcal{L} \subseteq \mathcal{P}$ : Menge aller aufsteigend sortierten Listen.
- ▶  $\mathcal{A}$ : Quicksort
- ▶  $\gamma(p)$ ,  $p \in \mathcal{P}$ : Länge von  $p$ , d.h. Anzahl Elemente in  $p$ .

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

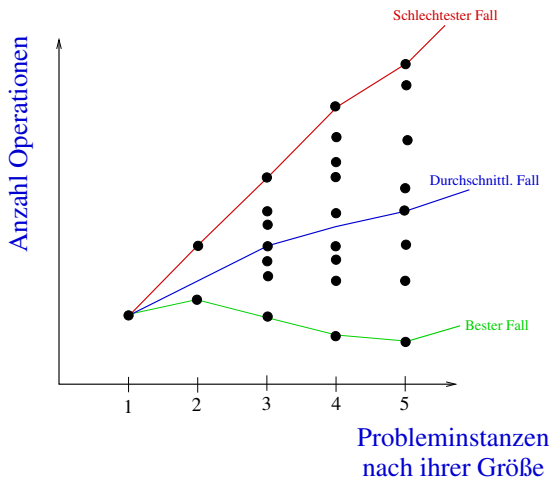
Kap. 10

Kap. 11  
656/170



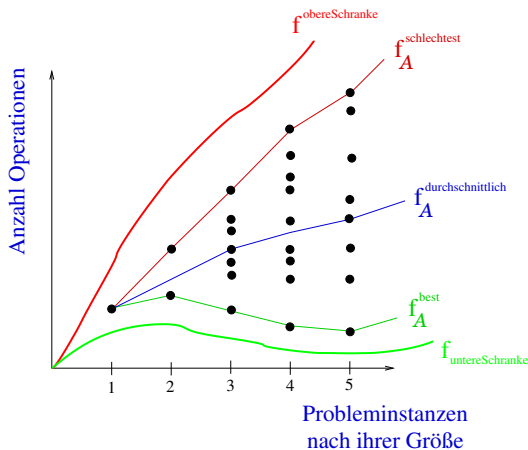
# Aufwand in Abhängigkeit der Problemgröße

...von Algorithmus  $\mathcal{A}$ , wobei  $\bullet$  Probleminstanzen darstellen:



# Aus Gründen der Praktikabilität

...betrachtet man i.a nicht die (meist schwer beschreibbaren) Aufwandsfunktionen  $f_A^{\text{schlechtest}}$ ,  $f_A^{\text{durchschnittlich}}$  und  $f_A^{\text{best}}$  direkt, sondern konzentriert sich auf Funktionen  $f^{\text{obereSchranke}}$  und  $f^{\text{untereSchranke}}$ , die  $f_A^{\text{schlechtest}}$  und  $f_A^{\text{best}}$  beschränken:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

658/170

# Obere, untere, einhüllende Schranken

Seien  $f, g : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$  zwei Funktionen von den natürlichen in die positiven reellen Zahlen (einschl. 0 jeweils; sonst  $\mathbb{N}$ ,  $\mathbb{R}^+$ ).

## Definition 7.4.1 (Obere, untere, einhüll. Schranke)

$g$  heißt **asymptotische**

- **obere Schranke** von  $f$  gdw.

$$\exists k \in \mathbb{R}^+. \exists n_0 \in \mathbb{N}_0. \forall n \in \mathbb{N}_0. n \geq n_0 \Rightarrow f(n) \leq k * g(n)$$

In diesem Fall schreiben wir:  $f \in O(g)$  (oder  $f = O(g)$ ).

- **untere Schranke** von  $f$  gdw.

$$\exists k \in \mathbb{R}^+. \exists n_0 \in \mathbb{N}_0. \forall n \in \mathbb{N}_0. n \geq n_0 \Rightarrow f(n) \geq k * g(n)$$

In diesem Fall schreiben wir:  $f \in \Omega(g)$  (oder  $f = \Omega(g)$ ).

- **einhüllende Schranke** von  $f$  gdw.  $g$  ist obere und untere Schranke von  $f$ :  $f \in O(g) \wedge f \in \Omega(g)$ .

In diesem Fall schreiben wir:  $f \in \Theta(g)$  (oder  $f = \Theta(g)$ ).

# Asymptotische einhüllende Schranke

Es gilt:

## Proposition 7.4.2

Folgende Aussagen sind logisch äquivalent:

1.  $g$  ist **einhüllende Schranke** von  $f$ .
2.  $\exists k, k' \in \mathbb{R}^+. \exists n_0, n'_0 \in \mathbb{N}_0.$   
 $(\forall n \in \mathbb{N}_0. n \geq n_0 \Rightarrow f(n) \leq k * g(n)) \wedge$   
 $(\forall n \in \mathbb{N}_0. n \geq n'_0 \Rightarrow f(n) \geq k' * g(n))$
3.  $\exists k, k' \in \mathbb{R}^+. \exists n_0 \in \mathbb{N}_0.$   
 $\forall n \in \mathbb{N}_0. n \geq n_0 \Rightarrow (f(n) \leq k * g(n) \wedge f(n) \geq k' * g(n))$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

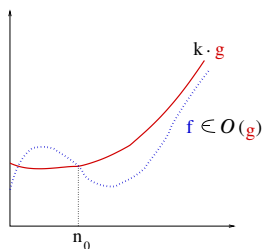
Kap. 9

Teil IV

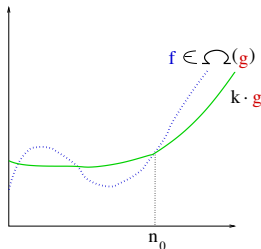
Kap. 10

Kap. 11

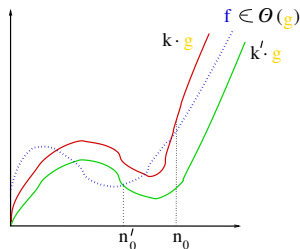
# Asymptotische Schranken: Veranschaulichung



Obere  
Schranke



Untere  
Schranke



Einhüllende  
Schranke

- ▶  $O(g) = \{f \mid g \text{ asypt. obere Schranke von } f\}$
- ▶  $\Omega(g) = \{f \mid g \text{ asypt. untere Schranke von } f\}$
- ▶  $\Theta(g) = \{f \mid g \text{ asypt. obere u. untere Schranke von } f\}$

# Sprechweisen (für $f, g : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ )

Man sagt, Funktion  $f$

► ist

- (a) höchstens
- (b) mindestens
- (c) genau

von der Größenordnung  $g$

► wächst von der Größenordnung (oder größenordnungs-  
mäßig)

- (a) höchstens
- (b) mindestens
- (c) genau

so schnell wie  $g$ , wenn gilt:

- (a)  $f \in O(g)$
- (b)  $f \in \Omega(g)$
- (c)  $f \in \Theta(g)$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Reflexivität, Transitivität, Symmetrie

## Lemma 7.4.3 (Reflexivität)

$$f \in O(f), f \in \Omega(f), f \in \Theta(f)$$

## Lemma 7.4.4 (Transitivität)

1.  $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$
2.  $f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h)$
3.  $f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$

## Lemma 7.4.5 (Symmetrie)

$$f \in \Theta(g) \iff g \in \Theta(f)$$

## Lemma 7.4.6 (Austauschsymmetrie)

$$f \in O(g) \iff g \in \Omega(f)$$

# Beachte

Die Schreibweise '=' für '∈', z.B.  $f = O(g)$  für  $f \in O(g)$  ist nützlich und verbreitet, aber ungenau, da '=' in diesem Kontext weder symmetrisch noch transitiv gelesen werden kann:

- ▶ Lesen wir  $f = O(g)$  gleichbedeutend mit  $f \in O(g)$ , so ist die Schreibweise  $O(g) = f$  sinnlos und ohne Bedeutung.
- ▶ Aus  $f = O(g)$  und  $h = O(g)$  kann nicht geschlossen werden:  $f = h$ .

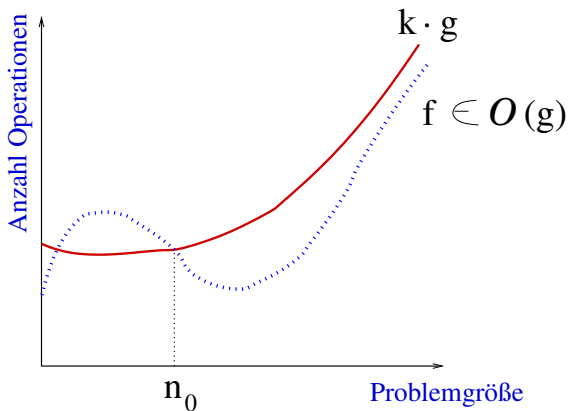
Das bedeutet:

- ▶ In Kontexten wie  $f = O(g)$ ,  $f = \Omega(g)$ ,  $f = \Theta(g)$  ist '=' als **Einweggleichung** ausschließlich und nicht umkehrbar von **links nach rechts** zu lesen.



# In der Praxis bes. wichtig: Asymp. obere Schr.

...sei  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  eine Funktion, die die max. Zahl elementarer Operationen eines Algorithmus  $\mathcal{A}$  in Abhängigkeit der durch eine natürliche Zahl beschriebenen Problemgröße angibt:



# Einige Rechengesetze für Groß-O (1)

Seien  $f, g : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$  zwei Funktionen von den natürlichen in die positiven reellen Zahlen und seien  $a, b \in \mathbb{R}_0^+$  zwei positive Konstanten.

## Lemma 7.4.7

1.  $a > 0 \Rightarrow O(af + b) = O(f)$
2.  $g \in O(f) \Rightarrow O(f + g) = O(f)$
3. Ist  $g$  streng monoton wachsend ( $\forall n, n' \in \mathbb{N}_0. n < n' \Rightarrow g(n) < g(n')$ ) oder monoton wachsend ( $\forall n, n' \in \mathbb{N}_0. n < n' \Rightarrow g(n) \leq g(n')$ ) und nicht beschränkt ( $\forall n \in \mathbb{N}_0. \exists n' \in \mathbb{N}_0. g(n') > n$ ), so gilt:

$$O(f) \subsetneq O(f \cdot g)$$

# Einige Rechengesetze für Groß-O (2)

## Lemma 7.4.8

Sind  $f, g$  polynomiale Funktionen mit

$$f(n) =_{df} \sum_{i=1}^k a_i \cdot n^i, \quad g(n) =_{df} \sum_{j=1}^l b_j \cdot n^j$$

wobei  $a_i, b_j \in \mathbb{R}_0^+$  für alle  $i, j$  und  $a_k, b_l > 0$ , so gilt:

1.  $O(f) = O(g) \Leftrightarrow k = l$
2.  $O(f) \subsetneq O(g) \Leftrightarrow k < l$

# Einige Rechengesetze für Groß-O (3)

Für alle Funktionen  $f, g : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$  gilt:

## Lemma 7.4.9 (Umkehrinklusion)

$$O(f) \subseteq O(g) \iff \Omega(f) \supseteq \Omega(g)$$

## Lemma 7.4.10 (Äquivalenzcharakterisierung)

Folgende Aussagen sind logisch äquivalent:

1.  $f \in \Theta(g)$
2.  $f \in O(g) \wedge f \in \Omega(g)$
3.  $\Theta(f) = \Theta(g)$
4.  $g \in O(f) \wedge g \in \Omega(f)$
5.  $g \in \Theta(f)$

## Lemma 7.4.11 (Quotientenfolgencharakterisierung)

Existiert  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$  mit  $0 < c \in \mathbb{R}^+$ , so gilt:  $f \in \Theta(g)$ .

# Einige Rechengesetze für Groß-O (4)

Aus Lemma 7.4.10 folgen als Korollare:

## Korollar 7.4.12 (Äquivalenzrelation)

Die Relation 'ist genau von der Größenordnung' ist eine Äquivalenzrelation (d.h. reflexive, transitive, antisymmetrische Relation) auf der Menge  $\mathcal{F} =_{df} [\mathbb{N}_0 \rightarrow \mathbb{R}_0^+]$  aller Funktionen von den natürlichen in die positiven reellen Zahlen.

Da Äquivalenzrelationen eine Partitionierung der Grundmenge induzieren (d.h. Äquivalenzklassen paarweise verschieden, Vereinigung aller Äquivalenzklassen gleich Grundmenge) und umgekehrt, erhalten wir weiters:

## Korollar 7.4.13 (Partitionierung)

Die Äquivalenzklassen der Relation 'ist genau von der Größenordnung' bilden eine Partitionierung von  $\mathcal{F} =_{df} [\mathbb{N}_0 \rightarrow \mathbb{R}_0^+]$ .

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
669/170

# Beispiele

...in der Praxis häufig auftretender **Kostenfunktionen**:

Kürzel	Aufwand	Intuition: <i>Vertausendfache Eingabe heißt...</i>
$O(c)$	konstant	gleiche Arbeit
$O(\log_2 n)$	logarithmisch	nur zehnfache Arbeit
$O(n)$	linear	auch vertausendfache Arbeit
$O(n \log_2 n)$	quasi-linear	zehntausendfache Arbeit
$O(n^2)$	quadratisch	millionenfache Arbeit
$O(n^3)$	kubisch	milliardenfache Arbeit
$O(n^c)$	polynomiell	gigantisch viel Arbeit (f. großes $c$ )
$O(2^n)$	exponentiell	hoffnungslos

Peter Pepper. **Funktionale Programmierung in OPAL, ML, Haskell und Gofer**. Springer-V., 2. Auflage, 2003, Kap. 11.

**Anm.:** Die Angabe der Basis bei (quasi-) logarithm. Komplexität entfällt üblicherw. (auch bei Pepper), da sie als Konstante in d.  $O$ -Not. aufgeht.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
670/170

# Veranschaulichung über Skalierbarkeit

...was das Wachstum von Problem instanzen für die reale Ausführungszeit bedeutet (Annahme: Jede Elementarop. benötigt  $1 \mu\text{s}$ ;  $f(n)$  Zahl max. benötigter Elementarop. für Instanzen d. Größe  $n$ ):

Grösse $n$	Linear $f(n) = n$	Quadratisch $f(n) = n^2$	Kubisch $f(n) = n^3$	Exponentiell $f(n) = 2^n$
1	$1 \mu\text{s}$	$1 \mu\text{s}$	$1 \mu\text{s}$	$2 \mu\text{s}$
10	$10 \mu\text{s}$	$100 \mu\text{s}$	1 ms	1 ms
20	$20 \mu\text{s}$	$400 \mu\text{s}$	8 ms	1 s
30	$30 \mu\text{s}$	$900 \mu\text{s}$	27 ms	18 min
40	$40 \mu\text{s}$	2 ms	64 ms	13 Tage
50	$50 \mu\text{s}$	3 ms	125 ms	36 Jahre
60	$60 \mu\text{s}$	4 ms	216 ms	36 560 Jahre
100	$100 \mu\text{s}$	10 ms	1 sec	$4 * 10^{16}$ Jahre
1000	1 ms	1 sec	17 min	sehr, sehr lange...

Peter Pepper. [Funktionale Programmierung in OPAL, ML, Haskell und Gofer](#). Springer-V., 2. Auflage, 2003, Kap. 11.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

671/170

# Veransch. über handhabbare Problemgröße (1)

...mit folgender **umgekehrter Fragestellung**:

- Probleminstanzen welcher Größe können **gerade noch gelöst** werden, so dass das Ergebnis **'geföhlt sofort'**, ohne merklich wahrnehmbare Berechnungsverzögerung vorliegt?

Dafür nehmen wir an:

- Ein Ergebnis liegt **'geföhlt sofort'** vor, wenn die Berechnungszeit  **$bz$**  nicht wesentlich mehr als eine **Hundertstel-sekunde** beträgt, d.h. wenn für die Berechnungszeit  **$bz(p)$**  einer Probleminstanz  **$p$**  gilt:

$$bz(p) < (10.000 + \varepsilon) \mu s = (10 + \varepsilon) ns = (0,01 + \varepsilon) s$$

- Eine Elementaroperation benötigt  **$1 \mu s$** .



# Veransch. über handhabbare Problemgröße (2)

Gegeben seien:

- ▶ Sechs unterschiedliche Lösungsalgorithmen mit
  - ▶ exponentieller ( $\mathcal{A}_1$ )  $f_{\mathcal{A}_1} \in O(2^n)$
  - ▶ kubischer ( $\mathcal{A}_2$ )  $f_{\mathcal{A}_2} \in O(n^3)$
  - ▶ quadratischer ( $\mathcal{A}_3$ )  $f_{\mathcal{A}_3} \in O(n^2)$
  - ▶ quasi-linearer ( $\mathcal{A}_4$ )  $f_{\mathcal{A}_4} \in O(n \log_2 n)$
  - ▶ linearer ( $\mathcal{A}_5$ )  $f_{\mathcal{A}_5} \in O(n)$
  - ▶ logarithmischer ( $\mathcal{A}_6$ )  $f_{\mathcal{A}_6} \in O(\log_2 n)$

Laufzeit.

- ▶ Funktionen  $f_{\mathcal{A}_i} : \mathbb{N} \rightarrow \mathbb{N}$ ,  $1 \leq i \leq 6$ , die für jeden Algorithmus angeben, wieviele Elementaroperationen  $\mathcal{A}_i$  für die Lösung einer Probleminstance der Größe  $n$ ,  $n \in \mathbb{N}$ , höchstens benötigt.

**Informell:**  $f_{\mathcal{A}_i}$  übersetzt Problemgröße in (bis auf konstanten Faktor) max. Anzahl der von  $\mathcal{A}_i$  benötigten Elementaroperationen zu je  $1 \mu s$ .

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
673/170

# Veransch. über handhabbare Problemgröße (3)

Algorithmus	Zeitbudget (10.000 + $\varepsilon$ ) $\mu$ s	Handhabbare Problem- größe im Zeitbudget
Exponentiell $f_{\mathcal{A}_1}(n) = 2^n$	$f_{\mathcal{A}_1}(13) = 8.192$ $f_{\mathcal{A}_1}(14) = 16.384$	14
Kubisch $f_{\mathcal{A}_2}(n) = n^3$	$f_{\mathcal{A}_2}(21) = 9.261$ $f_{\mathcal{A}_2}(22) = 10.648$ $f_{\mathcal{A}_2}(26) = 17.576$	26
Quadratisch $f_{\mathcal{A}_3}(n) = n^2$	$f_{\mathcal{A}_3}(100) = 10.000$	100
Quasi-linear $f_{\mathcal{A}_4}(n) = n \log_2 n$	$f_{\mathcal{A}_4}(1.000) \approx 10.000$	1.000
Linear $f_{\mathcal{A}_5}(n) = n$	$f_{\mathcal{A}_5}(10.000) = 10.000$	10.000
Logarithmisch $f_{\mathcal{A}_6}(n) = \log_2 n$	$f_{\mathcal{A}_6}(2^{10.000}) = 10.000$	$2^{10.000}$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
674/170

# Wachstum informell gedeutet

- **Exponentiell:** Die Zahl der Operationen wächst im Vergleich zur Problemgröße **äußerst schnell**:

Problemgröße $n$	Anzahl Operationen $2^n$
1	2
14	16.384
1.000	$2^{1.000}$
$n$	$2^n$

- **Logarithmisch:** Die Zahl der Operationen wächst im Vergleich zur Problemgröße **äußerst langsam**:

Problemgröße $2^n$	Anzahl Operationen $n$
1	0
$16 \approx 2^4$	4
$1.024 \approx 2^{10}$	10
$2^{16.384}$	16.384
$2^{2^n}$	$2^n$

# Lohnt der Kauf eines schnelleren Rechners?

Algorithmus	Alter Rechner	Neuer schnellerer Rechner		
		10x	100x	1.000x
Exponentiell $f_{A_1}(n) = 2^n$	14 ( $2^{14} = 16.384$ )	17 ( $2^{17} = 131.072$ )	20 ( $2^{20} = 1.048.576$ )	24 ( $2^{24} = 16.777.216$ )
Kubisch $f_{A_2}(n) = n^3$	26 ( $26^3 = 17.576$ )	47 ( $47^3 = 103.823$ )	100 ( $100^3 = 1.000.000$ )	216 ( $216^3 = 10.077.696$ )
Quadratisch $f_{A_3}(n) = n^2$	100 ( $100^2 = 10.000$ )	317 ( $317^2 = 100.489$ )	1.000 ( $1.000^2 = 1.000.000$ )	3.163 ( $3.163^2 = 10.004.569$ )
Quasi-linear $f_{A_4}(n) = n \log_2 n$	1.000 ( $10^3 \log_2 10^3$ $\approx 10^3 * 10$ $= 10.000$ )	9.000 ( $9.000 \log_2 9.000$ $\approx 9.000 * 13$ $= 117.000$ )	65.000 ( $65.000 \log_2 65.000$ $\approx 65.000 * 16$ $= 1.040.000$ )	530.000 ( $530.000 \log_2 530.000$ $\approx 530.000 * 19$ $= 10.070.000$ )
Linear $f_{A_5}(n) = n$	10.000	100.000	1.000.000	10.000.000
Logarithmisch $f_{A_6}(n) = \log_2 n$	$2^{10.000}$ ( $\log_2 2^{10^4}$ $= 10.000$ )	$2^{100.000}$ ( $\log_2 2^{10^5}$ $= 100.000$ )	$2^{1.000.000}$ ( $\log_2 2^{10^6}$ $= 1.000.000$ )	$2^{10.000.000}$ ( $\log_2 2^{10^7}$ $= 10.000.000$ )

# Beobachtung

Der Kauf eines neuen, **schnelleren Rechners** bringt verglichen mit dem Finden und Wechsel zu einem **asymptotisch besseren Algorithmus** fast **nichts**.

- ▶ **Exponentieller** Algorithmus, Rechner um **Faktor 1.000 schneller**, im Bsp. wächst handhabbar von **14** auf **24**:  
     $\rightsquigarrow$  **bedeutungslos**, weil noch immer weit zu wenig.
- ▶ **Logarithmischer** Algorithmus, Rechner um **Faktor 1.000 schneller**, im Bsp. wächst handhabbar von  **$2^{10.000}$**  auf  **$2^{10.000.000}$** :  
     $\rightsquigarrow$  **bedeutungslos**, weil schon  **$2^{10.000}$**  riesig (genug) ist.

**Kein Königsweg:**

- ▶ Das Finden eines asymptotisch besseren Algorithmus ist **kein Selbstläufer**; möglicherweise gibt es auch keinen.

# Interpretation, Folgerungen

- ▶ Einem asymptotisch schlechten Algorithmus hat auch ein schneller(er) Rechner nichts entgegenzusetzen:

Selbst ein 100.000-fach schnellerer Rechner mit 1.000.000.000 Operationen pro 0,01 s erlaubt bei exponentiellem Algorithmus ohne Verlängerung der erlaubten Antwortzeit lediglich Probleme bis zur Größe 30 zu lösen:  $2^{30} = 1.073.741.824$ .

- ▶ Ein asymp. schlechter Algorithmus verhält sich schlecht, auch wenn er auf einen schnelleren Rechner portiert wird.
- ▶ Asymp. schlechtes Alg.-Verhalten ist portierungsinvariant.

## Faustregel:

- ▶ Ein langsamer Rechner mit asymptotisch gutem Algorithmus gewinnt gegen einen schnellen Rechner mit asymptotisch schlechtem oder auch nur schlechterem Algorithmus bei Probleminstanzen nichttrivialer Größe immer!

# Abschließend

Die Beispiele und Überlegungen dieses Abschnitts machen deutlich:

- ▶ Rekursionsmuster beeinflussen (auch) die Effizienz einer Implementierung (siehe die naive baumartig-rekursive Implementierung der Fibonacci-Funktion).
- ▶ Die Wahl eines zweckmäßigen und zweckmäßig eingesetzten Rekursionsmusters ist deshalb wichtig.

**WICHTIG:** Nicht bestimmte Rekursionsmuster an sich sind

- ▶ problematisch, sondern ihr unzweckmäßiger Einsatz, wenn etwa wie im Fall der Fibonacci-Funktion baumartige Rekursion zu (unnötigen) Vielfachberechnungen von Werten führt!

Zweckmäßig eingesetzt bietet z.B. baumartige Rekursion viele Vorteile, darunter zur Parallelisierung! *Stichwort:* Teile und herrsche (oder divide et impera oder divide and conquer)!

# Übungsaufgabe 7.4.14

Vervollständige jede Spalte so weit bis d. Ausführungszeiten f. praktische Berechnungen gänzlich nutzlose Werte annehmen:

Grösse $n$	$f(n) = \log_{10} n$	$f(n) = \log_2 n$	$f(n) = n$	$f(n) = n \log_{10} n$	$f(n) = n \log_2 n$	$f(n) = n!$
1			1 $\mu s$			
10			10 $\mu s$			
20			20 $\mu s$			
30			30 $\mu s$			
40			40 $\mu s$			
50			50 $\mu s$			
60			60 $\mu s$			
100			100 $\mu s$			
1.000			1 ms			
10.000			10 ms			
100.000			100 ms			
1.000.000			1 s			

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

**7.4**

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
680/170



# Übungsaufgabe 7.4.15

Angenommen, ein Rechner einer neuen Generation benötigt für jede Elementaroperation nur noch  $1ns$  statt einer  $1\mu s$ .

1. Wie ändern sich die Werte in den beiden Tabellen aus [Abschnitt 7.4](#) und [Übungsaufgabe 7.4.14](#)?
2. Bis zu jeweils [welcher Problemgröße](#) können Algorithmen der verschiedenen Komplexitätsklassen unter praktischen Gesichtspunkten eingesetzt werden, d.h. wie lange sind wir willens, auf ein Ergebnis zu warten? (Am [Bsp.](#) der Tabelle aus [Abschnitt 7.4](#): 18 Minuten, vermutlich ja; 13 Tage, möglicherweise noch; 36 Jahre, eher nicht).
3. Ist für Algorithmen mit [logarithmischer Komplexität](#) ( $\log n$ ,  $n \log n$ ) der Unterschied zur [Basis 10](#) oder zur [Basis 2](#) unter praktischen Gesichtspunkten (d.h. im Sinn der vorherigen Teilaufgabe) [bedeutsam](#) oder [vernachlässigbar](#)?

# Übungsaufgabe 7.4.16 (1)

Manchmal, z.B. weil man keine asymptotische Schranke kennt oder findet, betrachtet man **starke** (oder **nicht-scharfe**) **asymptotische Schranken**.

Seien  $f, g : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$  zwei Funktionen von den natürlichen in die positiven reellen Zahlen.

## Definition 7.4.17 (Starke obere, untere Schranke)

$g$  heißt **starke** (oder **nicht-scharfe**) **asymptotische**

- **obere Schranke** von  $f$  gdw.

$$\forall k \in \mathbb{R}^+. \exists n_0 \in \mathbb{N}_0. \forall n \in \mathbb{N}_0. n \geq n_0 \Rightarrow f(n) \leq k * g(n)$$

In diesem Fall schreiben wir:  $f \in o(g)$  (oder  $f = o(g)$ ).

- **untere Schranke** von  $f$  gdw.

$$\forall k \in \mathbb{R}^+. \exists n_0 \in \mathbb{N}_0. \forall n \in \mathbb{N}_0. n \geq n_0 \Rightarrow f(n) \geq k * g(n)$$

In diesem Fall schreiben wir:  $f \in \omega(g)$  (oder  $f = \omega(g)$ ).

# Übungsaufgabe 7.4.16 (2)

## Lemma 7.4.18 (Quotientenfolgencharakterisierung)

1. Ist  $f \in o(g)$ , so gilt:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \quad (\wedge \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \infty \text{ wenn ex.})$$

2. Ist  $f \in \omega(g)$ , so gilt:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0 \quad (\wedge \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \text{ wenn ex.})$$

### Informell:

- ▶ Gilt  $f \in o(g)$ , so sind für große  $n$  die Werte von  $f(n)$  unbedeutend gegenüber denen von  $g(n)$ .
- ▶ Gilt  $f \in \omega(g)$ , so sind für große  $n$  die Werte von  $g(n)$  unbedeutend gegenüber denen von  $f(n)$ .

# Übungsaufgabe 7.4.16 (3)

Welche folgender Aussagen gelten? Beweis oder Gegenbeispiel.

## 1. Reflexivität:

1.1  $f \in o(f)$

1.2  $f \in \omega(f)$

## 2. Transitivität:

2.1  $f \in o(g) \wedge g \in o(h) \Rightarrow f \in o(h)$

2.2  $f \in \omega(g) \wedge g \in \omega(h) \Rightarrow f \in \omega(h)$

## 3. Symmetrie:

3.1  $f \in o(g) \iff g \in o(f)$

3.2  $f \in \omega(g) \iff g \in \omega(f)$

4. Austauschsymmetrie:  $f \in o(g) \iff g \in \omega(f)$

5. Umkehrinklusion:  $o(f) \subseteq o(g) \iff \omega(f) \supseteq \omega(g)$

## 6. Schnitt:

6.1  $O(g) \cap \omega(g) = \emptyset$

6.2  $o(g) \cap \Omega(g) = \emptyset$

6.3  $o(g) \cap \omega(g) = \emptyset$

# Übungsaufgabe 7.4.16 (4)

Gilt  $f \in O(g)$  ( $f \in \Omega(g)$ ), so kann  $g$  asymptotisch scharfe obere (untere) Schranke von  $f$  sein oder nicht.

Gib je ein Beispiel für Funktionen  $f$  und  $g$  an, so dass gilt:

1.  $f \in O(g)$  und  $g$  ist

1.1 scharfe

1.2 nicht-scharfe

asymptotische obere Schranke von  $f$ .

2.  $f \in \Omega(g)$  und  $g$  ist

2.1 scharfe

2.2 nicht-scharfe

asymptotische untere Schranke von  $f$ .

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
685/170

# Landausche Symbole

Die Symbole  $O$ ,  $\Omega$ ,  $\Theta$ ,  $o$  und  $\omega$  heißen **Landausche Symbole** und gehen zurück auf Arbeiten von:

- ▶ Edmund Landau. **Handbuch der Lehre von der Verteilung der Primzahlen**, Band I und II, B. G. Teubner, 1909. ( $o$ -Notation)
- ▶ Paul Bachmann. **Die Analytische Zahlentheorie**. Zahlentheorie, B. G. Teubner, 1894. ( $O$ -Notation)
- ▶ Godfrey H. Hardy, John E. Littlewood. **Some Problems of Diophantine Approximation**. Acta Mathematica 37:155-238, 1914. ( $\Omega$ -Notation; mit vom heutigen Gebrauch abweichender Bedeutung von  $\Omega$  als 'o-Negation')

Für einen genaueren historischen Abriss siehe die Arbeit:

- ▶ Donald E. Knuth. **Big Omicron and Big Omega and Big Theta**. ACM SIGACT News 8(2):18-24, 1976.

...der für die Verwendung der Landauschen Symbole in ihrer heutigen Bedeutung maßgeblicher Einfluss zukommt.

# Kapitel 7.5

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

**7.5**

Kap. 8




Kap. 9

Teil IV

Kap. 10

Kap. 11

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 7 (1)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 4, Rekursion als Entwurfstechnik; Kapitel 9, Laufzeitanalyse von Algorithmen; Kapitel 9.2, Landau-Symbole)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 11, Software-Komplexität)
-  Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein. *Algorithmen – Eine Einführung*. Oldenbourg Verlag, 2004. Kapitel 3, Wachstum von Funktionen; Kapitel 3.1, Asymptotische Notation ( $\Theta$ ,  $O$ ,  $\Omega$ ,  $o$ ,  $\omega$ ); Kapitel 3.2, Standardnotationen und Standardfunktionen)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9




Teil IV

Kap. 10

Kap. 11



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 7 (2)

-  Ben Goldreich. *Invitation to Complexity Theory*. Crossroads, the ACM Magazine for Students 18(3):18-22, 2012.
-  Neil D. Jones. *Constant Time Factors do Matter*. In Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC'93), 602-611, 1993.
-  Donald E. Knuth. *Big Omicron and Big Omega and Big Theta*. ACM SIGACT News 8(2):18-24, 1976.  
(s.a. Nachdruck unter gleichem Titel in: Donald E. Knuth. *Selected Papers on Analysis of Algorithms*. CSLI Lecture Notes Number 102, CSLI Publications, 35-41, 2012.)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 7 (3)

-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 5, Rekursion; Kapitel 11, Formalismen 3: Aufwand und Terminierung)
-  Steven S. Skiena. *The Algorithm Design Manual*. Springer-V., 1998. (Kapitel 1.3.2, Best, Worst, and Average-Case Complexity; Kapitel 1.4, The Big Oh Notation)
-  Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik. Induktives Vorgehen*. Springer-V., 2014. (Kapitel 4.1.3, Induktiv definierte Algorithmen. Türme von Hanoi)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

7.1

7.2

7.3

7.4

7.5

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11  
690/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 7 (4)

-  Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018. (Kapitel 4.1.3, Inductively defined Algorithms. Towers of Hanoi)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 20, Time and space behaviour)
-  Ingo Wegener. *Komplexität*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 119-144, 2006. (Kapitel 5.1, Größenordnungen und die  $\mathcal{O}$ -Notation)

# Kapitel 8

## Auswertung von Ausdrücken

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

**Kap. 8**

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

# Auswertung

...einfacher und einfacher funktionaler Ausdrücke.

Zentral: Die Organisation des Zusammenspiels von

- ▶ **Expandieren** ( $\rightsquigarrow$  Funktionsaufrufe)
- ▶ **Simplifizieren** ( $\rightsquigarrow$  einfache Ausdrücke)

um einen Ausdruck soweit zu vereinfachen wie möglich.

# Kapitel 8.1

## Auswertung einfacher Ausdrücke

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

-694/170

# Auswerten einfacher Ausdrücke

Viele (**Simplifikations-**) Wege führen zum (selben!) Ziel, hier zum Wert **42**, der **Semantik** (oder **Bedeutung**) des Ausdrucks  $3*(9+5)$ :

**Simplifikations-Weg 1:**

$$\begin{aligned} & 3 * (9+5) \\ (S) \rightarrow & 3 * 14 \\ (S) \rightarrow & 42 \end{aligned}$$

**S-Weg 2:**

$$\begin{aligned} & 3 * (9+5) \\ (S) \rightarrow & 3*9 + 3*5 \\ (S) \rightarrow & 27 + 3*5 \\ (S) \rightarrow & 27 + 15 \\ (S) \rightarrow & 42 \end{aligned}$$

**S-Weg 3:**

$$\begin{aligned} & 3 * (9+5) \\ (S) \rightarrow & 3*9 + 3*5 \\ (S) \rightarrow & 3*9 + 15 \\ (S) \rightarrow & 27 + 15 \\ (S) \rightarrow & 42 \end{aligned}$$

# Kapitel 8.2

## Auswertung einfacher funktionaler Ausdrücke

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

**8.2**

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

-696/170



# Bsp. 1: Auswerten von Funktionsaufrufen

```
simple x y z :: Int -> Int -> Int -> Int
```

```
simple x y z = (x + z) * (y + z)
```

Der Ausdruck `simple 2 3 4` hat als **Semantik** (oder **Bedeutung**) den Wert **42**:

**ES-Weg 1:**                      `simple 2 3 4`  
(Expandieren)  $\rightarrow$   $(2 + 4) * (3 + 4)$   
(Simplifizieren)  $\rightarrow$   $6 * (3 + 4)$   
(S)  $\rightarrow$   $6 * 7$   
(S)  $\rightarrow$  **42**

**ES-Weg 2:**                      `simple 2 3 4`  
(E)  $\rightarrow$   $(2 + 4) * (3 + 4)$   
(S)  $\rightarrow$   $(2 + 4) * 7$   
(S)  $\rightarrow$   $6 * 7$   
(S)  $\rightarrow$  **42**

...und viele weitere **ES-Wege**; alle zur selben **Bedeutung**: **42**.

## Bsp. 2: Auswerten von Funktionsaufrufen

```
zip :: [a] -> [b] -> [(a,b)]
zip _ []           = []
zip [] _          = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Der Ausdruck `zip [1,3,5] [2,4,6,8,10]` hat als **Semantik** (oder **Bedeutung**) den Wert `[(1,2),(3,4),(5,6)]`:

```
zip [1,3,5] [2,4,6,8,10]
-- Syntakt. Zucker der Listennotation auflösen
->> zip (1:(3:(5:[]))) (2:(4:(6:(8:(10:[]))))))
(E/S) ->> (1,2) : zip (3:(5:[])) (4:(6:(8:(10:[]))))
(E/S) ->> (1,2) : ((3,4) : zip (5:[]) (6:(8:(10:[]))))
(E/S) ->> (1,2) : ((3,4) : ((5,6) : zip [] (8:(10:[]))))
(E/S) ->> (1,2) : ((3,4) : ((5,6) : []))
-- Syntakt. Zucker für Listennotation einführen
->> [(1,2),(3,4),(5,6)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

698/1700

# Bsp. 3: Auswerten von Funktionsaufrufen (1)

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

Der Ausdruck `fac 2` hat als **Semantik** (oder **Bedeutung**) den Wert **2**:

```
fac 2
(Expandieren) ->> if 2 == 0 then 1
                  else (2 * fac (2 - 1))
(Simplifizieren) ->> 2 * fac (2 - 1)
```

Für die **Fortführung** der **Berechnung**

- gibt es **Freiheitsgrade** und damit verschiedene Möglichkeiten.

**Zwei** dieser **Möglichkeiten** führen wir in der Folge genauer aus.

## Bsp. 3: Auswerten von Funktionsaufrufen (2)

### Variante a)

```
2 * fac (2 - 1)
(Simplifizieren) ->> 2 * fac 1
(Expandieren) ->> 2 * (if 1 == 0 then 1
                      else (1 * fac (1-1)))
->> ... in diesem Stil fortfahren
```

### Variante b)

```
2 * fac (2 - 1)
(Expandieren) ->> 2 * (if (2-1) == 0 then 1
                      else ((2-1) * fac ((2-1)-1)))
(Simplifizieren) ->> 2 * ((2-1) * fac ((2-1)-1))
->> ... in diesem Stil fortfahren
```

## Bsp. 3: Auswertung gemäß Variante a)

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

```
    fac 2
```

```
(E) ->> if 2 == 0 then 1 else (2 * fac (2 - 1))
```

```
(S) ->> if False then 1 else (2 * fac (2 - 1))
```

```
(S) ->> (2 * fac (2 - 1))
```

```
(S) ->> 2 * fac 1
```

```
(E) ->> 2 * (if 1 == 0 then 1 else (1 * fac (1 - 1)))
```

```
(S) ->> 2 * (if False then 1 else (1 * fac (1 - 1)))
```

```
(S) ->> 2 * ((1 * fac (1 - 1)))
```

```
(S) ->> 2 * (1 * fac 0)
```

```
(E) ->> 2 * (1 * (if 0 == 0 then 1  
                  else (0 * fac (0 - 1))))
```

```
(S) ->> 2 * (1 * (if True then 1  
                  else (0 * fac (0 - 1))))
```

```
(S) ->> 2 * (1 * 1)
```

```
(S) ->> 2 * 1
```

```
(S) ->> 2
```

⇝ sog. **applikativer** Auswertungstil.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

701/170

## Bsp. 3: Auswertung gemäß Variante b)

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

```
    fac 2
```

```
(E) ->> if 2 == 0 then 1 else (2 * fac (2 - 1))
```

```
(S) ->> if False then 1 else (2 * fac (2 - 1))
```

```
(S) ->> (2 * fac (2 - 1))
```

```
(E) ->> 2 * (if (2-1) == 0 then 1  
              else ((2-1) * fac ((2-1)-1)))
```

```
(2S) ->> 2 * (if False then 1  
              else ((2-1) * fac ((2-1)-1)))
```

```
(S) ->> 2 * (((2-1) * fac ((2-1)-1)))
```

```
(S) ->> 2 * (1 * fac ((2-1)-1))
```

```
(E) ->> 2 * (1 * (if ((2-1)-1) == 0 then 1  
                  else ((2-1)-1) * fac (((2-1)-1)-1)))
```

```
(E) ->> 2 * (1 * (if True then 1  
                  else ((2-1)-1) * fac (((2-1)-1)-1)))
```

```
(3S) ->> 2 * (1 * 1)
```

```
(2S) ->> 2
```

↪ sog. **normaler** Auswertungsstil.

## Bsp. 3: Applikative Auswertung von fac 3

fac 3

```
(E) ->> if 3 == 0 then 1 else (3 * fac (3-1))
(S) ->> if False then 1 else (3 * fac (3-1))
(S) ->> (3 * fac (3-1))
(S) ->> 3 * fac 2
(E) ->> 3 * (if 2 == 0 then 1 else (2 * fac (2-1)))
(S) ->> 3 * (if False then 1 else (2 * fac (2-1)))
(S) ->> 3 * ((2 * fac (2-1)))
(S) ->> 3 * (2 * fac 1)
(E) ->> 3 * (2 * (if 1 == 0 then 1 else (1 * fac (1-1))))
(S) ->> 3 * (2 * (if False then 1 else (1 * fac (1-1))))
(S) ->> 3 * (2 * ((1 * fac (1-1))))
(S) ->> 3 * (2 * (1 * fac 0))
(E) ->> 3 * (2 * (1 * (if 0 == 0 then 1 else (0 * fac (0-1)))))
(S) ->> 3 * (2 * (1 * (if True then 1 else (0 * fac (0-1)))))
(S) ->> 3 * (2 * (1 * (1)))
(S) ->> 3 * (2 * 1)
(S) ->> 3 * 2
(S) ->> 6
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

703/170

## Bsp. 3: Normale Auswertung von fac 3 (1)

fac 3

```
(E) ->> if 3 == 0 then 1 else (3 * fac (3-1))
(S) ->> if False then 1 else (3 * fac (3-1))
(S) ->> (3 * fac (3-1))
(E) ->> 3 * (if (3-1) == 0 then 1 else ((3-1) * fac ((3-1)-1)))
(S) ->> 3 * (if 2 == 0 then 1 else ((3-1) * fac ((3-1)-1)))
(S) ->> 3 * (if False then 1 else ((3-1) * fac ((3-1)-1)))
(S) ->> 3 * (((3-1) * fac ((3-1)-1)))
(S) ->> 3 * (2 * fac ((3-1)-1))
(E) ->> 3 * (2 * (if ((3-1)-1) == 0 then 1
                  else (((3-1)-1) * fac (((3-1)-1)-1))))
(S) ->> 3 * (2 * (if (2-1) == 0 then 1
                  else (((3-1)-1) * fac (((3-1)-1)-1))))
(S) ->> 3 * (2 * (if 1 == 0 then 1
                  else (((3-1)-1) * fac (((3-1)-1)-1))))
(S) ->> 3 * (2 * (if False then 1
                  else (((3-1)-1) * fac (((3-1)-1)-1))))
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

704/170



## Bsp. 3: Normale Auswertung von fac 3 (2)

```
(S) ->> 3 * (2 * (((3-1)-1) * fac (((3-1)-1)-1)))
(S) ->> 3 * (2 * ((2-1) * fac (((3-1)-1)-1)))
(S) ->> 3 * (2 * (1 * fac (((3-1)-1)-1)))
(E) ->> 3 * (2 * (1 *
    (if (((3-1)-1)-1) == 0 then 1
        else (((3-1)-1)-1) * fac (((3-1)-1)-1))))))
(S) ->> 3 * (2 * (1 *
    (if ((2-1)-1) == 0 then 1
        else (((3-1)-1)-1) * fac (((3-1)-1)-1))))))
(S) ->> 3 * (2 * (1 *
    (if (1-1) == 0 then 1
        else (((3-1)-1)-1) * fac (((3-1)-1)-1))))))
(S) ->> 3 * (2 * (1 *
    (if 0 == 0 then 1
        else (((3-1)-1)-1) * fac (((3-1)-1)-1))))))
(S) ->> 3 * (2 * (1 *
    (if True then 1
        else (((3-1)-1)-1) * fac (((3-1)-1)-1))))))
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

705/170

## Bsp. 3: Normale Auswertung von fac 3 (3)

(S) ->> 3 \* (2 \* (1 \* (1)))

(S) ->> 3 \* (2 \* (1 \* 1))

(S) ->> 3 \* (2 \* 1)

(S) ->> 3 \* 2

(S) ->> 6

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

**8.2**

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

706/1700

## Bsp. 4: Applikative Auswertung von natSum 3

```
natSum n = if n == 0 then 0 else natSum (n-1) + n
```

```
natSum 3
```

```
(E) ->> if 3 == 0 then 0 else natSum (3-1) + 3
```

```
(S) ->> if False then 0 else natSum (3-1) + 3
```

```
(S) ->> natSum (3-1) + 3
```

```
(S) ->> natSum 2 + 3
```

```
(E) ->> (if 2 == 0 then 0 else natSum (2-1) + 2) + 3
```

```
(S) ->> (if False then 0 else natSum (2-1) + 2) + 3
```

```
(S) ->> (natSum (2-1) + 2) + 3
```

```
(S) ->> (natSum 1 + 2) + 3
```

```
(E) ->> ...
```

```
...
```

```
(S) ->> 6
```

Übungsaufgabe 8.2.1: Vervollständige die obige Auswertung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

707/170

## Bsp. 4: Normale Auswertung von natSum 3

```
natSum n = if n == 0 then 0 else natSum (n-1) + n
```

```
natSum 3
```

```
(E) ->> if 3 == 0 then 0 else natSum (3-1) + 3
```

```
(S) ->> if False then 0 else natSum (3-1) + 3
```

```
(S) ->> natSum (3-1) + 3
```

```
(E) ->> (if (3-1) == 0 then 0 else natSum ((3-1)-1) + (3-1)) + 3
```

```
(S) ->> (if 2 == 0 then 0 else natSum ((3-1)-1) + (3-1)) + 3
```

```
(S) ->> (if False then 0 else natSum ((3-1)-1) + (3-1)) + 3
```

```
(S) ->> natSum ((3-1)-1) + (3-1) + 3
```

```
(E) ->> (if ((3-1)-1) == 0 then 0 else ... ) + (3-1) + 3
```

```
...
```

```
(S) ->> 6
```

Übungsaufgabe 8.2.2: Vervollständige die obige Auswertung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

708/170

## Ü-Aufgabe 8.2.3: Vervollst. d. applikat. Ausw.

```
natSum n
| n == 0    = 0
| otherwise = natSum (n-1) + n
```

```
    natSum 3
```

```
(E) ->> | 3 == 0    = 0
        | otherwise = natSum (3-1) + 3
```

```
(S) ->> | False = 0
        | True  = natSum (3-1) + 3
```

```
(S) ->> natSum (3-1) + 3
```

```
(S) ->> natSum 2 + 3
```

```
(E) ->> | 2 == 0    = 0
        | otherwise = natSum (2-1) + 2 + 3
```

```
(S) ->> | False = 0
        | True  = natSum (2-1) + 2 + 3
```

```
(S) ->> natSum (2-1) + 2 + 3
```

```
(S) ->> natSum 1 + 2 + 3
```

```
(E) ->> ...
```

```
    ...
```

```
(S) ->> 6
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

709/170

## Ü-Aufgabe 8.2.4: Vervollst. d. normale Ausw.

```
natSum n
| n == 0      = 0
| otherwise = natSum (n-1) + n
```

```
      natSum 3
(E) ->> | 3 == 0      = 0
        | otherwise = natSum (3-1) + 3
(S) ->> | False = 0
        | True  = natSum (3-1) + 3
(S) ->> natSum (3-1) + 3
(E) ->> | (3-1) == 0 = 0
        | otherwise = natSum ((3-1)-1) + (3-1) + 3
(S) ->> | False = 0
        | True  = natSum ((3-1)-1) + (3-1) + 3
(S) ->> natSum ((3-1)-1) + (3-1) + 3
(E) ->> | ((3-1)-1) == 0 = 0
        | otherwise      = ...
      ...
(S) ->> 6
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

710/170

# Übungsaufgabe 8.2.5

Betrachte die Deklaration der Funktion `fac`:

```
fac n
| n == 0      = 1
| otherwise = n * fac (n-1)
```

Werte den Aufruf `fac 3`

1. applikativ
2. nomal

aus.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

711/170

## Übungsaufgabe 8.2.6

Betrachte die Deklaration der Funktionen `fun91` und `fun91'`:

```
fun91 :: Integer -> Integer
```

```
fun91 n
```

```
  | n > 100 = n - 10
```

```
  | n <= 100 = fun91 (fun91 (n+11))
```

```
fun91' :: Integer -> Integer
```

```
fun91' n =
```

```
  if n > 100 then n - 10 else fun91' (fun91' (n+11))
```

Werte die Aufrufe `(fun91 101)` und `(fun91 100)` sowie `(fun91' 101)` und `(fun91' 100)` für jeweils eine Handvoll Expansionsschritte

1. applikativ

2. nominal

aus.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

712/1700



# Übungsaufgabe 8.2.7

Betrachte noch einmal die Deklaration der Funktion `zip` aus [Beispiel 2](#):

```
zip :: [a] -> [b] -> [(a,b)]
zip _ []           = []
zip [] _          = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Löse die in [Beispiel 2](#) kombinierten **Expansions-/Simplifikations**-Schritte in Einzelschritte auf, wobei  $\pi(\cdot, \cdot)$  für Musterpassung stehe:

```
      zip (1:(3:(5:[]))) (2:(4:(6:(8:(10:[]))))))
(E) ->> |  $\pi(1:(3:(5:[])), \_)$   $\pi(2:(4:(6:(8:(10:[]))))), []$  = []
      |  $\pi(1:(3:(5:[])), [])$   $\pi(2:(4:(6:(8:(10:[]))))), \_$  = []
      |  $\pi(1:(3:(5:[])), x:xs)$   $\pi(2:(4:(6:(8:(10:[]))))), y:ys$ 
      = (x,y) : zip xs ys
(S) ->> | Failed = []
      | Failed = []
      | Matched = (1,2) : zip (3:(5:[])) (4:(6:(8:(10:[]))))
(S) ->> (1,2) : zip (3:(5:[])) (4:(6:(8:(10:[]))))
      ->> ... ->> (1,2) : ((3,4) : ((5,6) : []))
```

Vervollständige die begonnene Auswertung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

713/170

# Kapitel 8.3

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

**8.3**

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

714/170

# Rückblick

...in Kapitel 8.1 und 8.2 haben wir anhand von Beispielen die Auswertung

- ▶ einfacher Ausdrücke (ohne Funktionsaufrufe)
- ▶ isolierter Funktionsaufrufe (ohne Einbettung in komplexe Ausdrücke) im
  - ▶ applikativen
  - ▶ normalen

Auswertungsstil

als maximale (d.h. nicht mehr verlängerbare) Folgen von Simplifikations- und Expansions-Schritten betrachtet.

Für alle Beispiele hat gegolten, dass die konkret gewählte Folge von Expansions- und Simplifikations-Schritten keinen Einfluss auf den Wert des jeweiligen Ausdrucks gehabt hat.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

715/170

# Ausblick

In Kapitel 12 und 13 werden wir zeigen, dass dies kein Zufall ist, sondern stets gilt, auch wenn z.B.

- ▶ Funktionsaufrufe in komplexe Ausdrücke eingebettet sind
- ▶ applikativer und normaler Auswertungsstil gemischt werden
- ▶ in irgendeiner, weder applikativen noch normalen, terminierenden Reihenfolge ausgewertet wird.

Dieses für die Wohldefinierbarkeit der Semantik funktionaler Programmiersprachen zentrale Resultat geht auf Alonzo Church und John Barkley Rosser zurück, das wir hier im Vorgriff auf Kap. 12.3.4 und Kap. 13 anführen:

## Theorem 8.4.1 (Church/Rosser, 1936)

Jede maximale terminierende Folge von Expansions- und Simplifikations-Schritten endet mit demselben Wert.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

716/170

# Kapitel 8.4

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

**8.4**

Kap. 9




Teil IV

Kap. 10

Kap. 11

717/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 8 (1)

-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 1, Problem Solving, Programming, and Calculation)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Kapitel 1, Introduction)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 9, Formalismen 1: Zur Semantik von Funktionen)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

718/170

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 8 (2)



Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999.  
(Kapitel 1, Introducing functional programming)



Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.  
(Kapitel 1, Introducing functional programming)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

8.1

8.2

8.3

8.4

Kap. 9

Teil IV

Kap. 10

Kap. 11

719/170

# Kapitel 9

## Programmentwicklung, Programmverstehen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

**Kap. 9**

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V



# Kapitel 9.1

## Programmentwicklung

Exercitatio artem parat.  
Übung verschafft Geschicklichkeit.  
Tacitus (um 55 - um 120 n.Chr.)  
röm. Geschichtsschreiber

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

721/170

# Systematischer Programmentwurf

Grundsätzlich gilt:

- ▶ Das Finden eines algorithmischen Lösungsverfahrens
  - ▶ ist ein kreativer Prozess
  - ▶ kann (deshalb) nicht vollständig automatisiert werden

Dennoch gibt es

- ▶ Vorgehensweisen und Faustregeln

die die Aussicht, erfolgreich zu sein, erhöhen.

Eine Vorgehensweise für die

- ▶ systematische Entwicklung rekursiver Programme

wollen wir in der Folge betrachten.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

722/170

# Systematische Programmentwicklung

...für **rekursive** Programme in einem **5-schrittigen** Prozess.

**5-schrittiger Entwurfsprozess** (Graham Hutton, 2007):

1. Lege die (Daten-) Typen fest.
2. Führe alle relevanten Fälle auf.
3. Lege die Lösung für die einfachen (Basis-) Fälle fest.
4. Lege die Lösung für die übrigen Fälle fest.
5. Verallgemeinere und vereinfache das Lösungsverfahren.

Dieses Vorgehen werden wir an **drei Beispielen** demonstrieren.

**Repetitio est mater studiorum.**  
**Wiederholung ist die Mutter der Studien.**

lat., sprichwörtl.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

723/170

# Bsp. 1: Aufsummieren

...der Elemente einer Liste ganzer Zahlen.

- Schritt 1: Lege die (Daten-) Typen fest

```
sum :: [Integer] -> Integer
```

- Schritt 2: Führe alle relevanten Fälle auf

```
sum [] =
```

```
sum (n:ns) =
```

- Schritt 3: Lege die Lösung für die Basisfälle fest

```
sum [] = 0
```

```
sum (n:ns) =
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

724/1700

# Bsp. 1: Aufsummieren (fgs.)

- Schritt 4: Lege die Lösung für die übrigen Fälle fest

```
sum []      = 0
sum (n:ns) = n + sum ns
```

- Schritt 5: Verallgemeinere u. vereinfache das Lösungsverf.

```
5a) sum :: Num a => [a] -> a
5b) sum = foldr (+) 0
```

## Gesamtlösung nach Schritt 5:

```
sum :: Num a => [a] -> a
sum = foldr (+) 0
```

# Bsp. 2: Streichen

...der ersten  $n$  Elemente einer Liste beliebigen Elementtyps.

- **Schritt 1:** Lege die (Daten-) Typen fest

`drop :: Int -> [a] -> [a]`

- **Schritt 2:** Führe alle relevanten Fälle auf

`drop 0 [] =`

`drop 0 (x:xs) =`

`drop (n+1) [] =`

`drop (n+1) (x:xs) =`

- **Schritt 3:** Lege die Lösung für die Basisfälle fest

`drop 0 [] = []`

`drop 0 (x:xs) = x:xs`

`drop (n+1) [] = []`

`drop (n+1) (x:xs) =`

## Bsp. 2: Streichen (fgs.)

- Schritt 4: Lege die Lösung für die übrigen Fälle fest

```
drop 0 []           = []
drop 0 (x:xs)       = x:xs
drop (n+1) []       = []
drop (n+1) (x:xs) = drop n xs
```

- Schritt 5: Verallgemeinere u. vereinfache das Lösungsv.

5a)  $\text{drop} :: \text{Integral } b \Rightarrow b \rightarrow [a] \rightarrow [a]$

```
5b) drop 0 xs           = xs
    drop (n+1) []       = []
    drop (n+1) (x:xs) = drop n xs
```

```
5c) drop 0 xs           = xs
    drop _ []           = []
    drop (n+1) (_,xs) = drop n xs
```

## Bsp. 2: Streichen (fgs.)

### Gesamtlösung nach Schritt 5:

```
drop :: Integral b => b -> [a] -> [a]
drop 0 xs          = xs
drop _ []          = []
drop (n+1) (_:xs) = drop n xs
```

### Hinweis:

- Muster der Form  $(n+1)$  werden von neueren Haskell-Versionen nicht mehr unterstützt. Deshalb:

```
drop :: Integral b => b -> [a] -> [a]
drop 0 xs          = xs
drop _ []          = []
drop n (_:xs) = drop (n-1) xs
```



# Bsp. 3: Entfernen

... des letzten Elements einer nichtleeren Liste beliebigen Elementtyps.

- Schritt 1: Lege die (Daten-) Typen fest

```
rmLast :: [a] -> [a]
```

- Schritt 2: Führe alle relevanten Fälle auf

```
rmLast (x:xs) =
```

- Schritt 3: Lege die Lösung für die Basisfälle fest

```
rmLast (x:xs) | null xs    = []  
               | otherwise =
```

## Bsp. 3: Entfernen (fgs.)

- Schritt 4: Lege die Lösung für die übrigen Fälle fest

```
rmLast (x:xs) | null xs    = []  
              | otherwise = x : rmLast xs
```

- Schritt 5: Verallgemeinere u. vereinfache das Lösungsverf.

5a) `rmLast :: [a] -> [a] -- keine Verallg. möegl.`

```
5b) rmLast []      = []  
    rmLast (x:xs) = x : rmLast xs
```

### Gesamtlösung nach Schritt 5:

```
rmLast :: [a] -> [a]  
rmLast []      = []  
rmLast (x:xs) = x : rmLast xs
```

# Verfeinerter Entwurfsprozess

Norman Ramsey (2014) schlägt einen vergleichbaren 7- bzw. 8-schrittigen Entwurfsprozess vor, der einen Vorschlag von Matthias Felleisen et al. (2001) verfeinert:

1. A.&B. Beschreibe die Daten, die die Funktion benutzt.
2. Beschreibe mithilfe der Signatur, einer Kopfzeile und eine Aufgabenbeschreibung, was die Funktion leistet.
3. Gib Beispiele an, die veranschaulichen und zeigen, was die Funktion leistet.
4. Schreibe ein Skelett (eine Definition mit noch auszufüllenden Lücken) der Funktion (engl. [template](#)).
5. Vervollständige das Skelett zu einer vollständigen Funktionsimplementierung (engl. [code](#)).
6. Teste die Funktion.
7. Beurteile die Funktion und refaktorisiere sie bei Bedarf.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

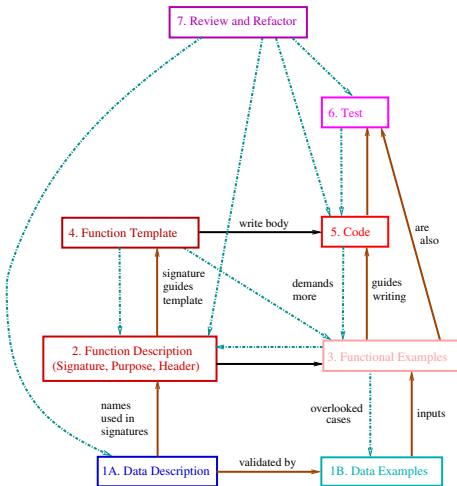
Kap. 11

Teil V

731/1700

# Graphische Darstellung

...des Entwurfsprozesses nach Ramsey:



Solid arrows: show initial design  
Dotted arrows: show feedback

Norman Ramsey.  
On Teaching How to Design Programs.  
In Proceedings ICFP 2014, Figure 1, p. 154.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

732/170

# Kapitel 9.2

## Programmverstehen

Exercitatio optimus magister.  
Übung ist der beste Lehrmeister.  
lat., sprichwörtl.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

**9.2**

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

733/170

# Motivation

...eine **Binsenweisheit**:

- ▶ Programme werden **häufiger gelesen** als **geschrieben**!

Deshalb ist es wichtig

- ▶ **Strategien**

zu besitzen, die durch geeignete **Vorgehensweisen** und **Fragen** an ein Programm helfen, Programme

- ▶ **zu lesen** und **zu verstehen**, besonders **fremde Programme**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

# Vier Vorgehensweisen

...im Überblick:

- 1) Lesen des Programmtexts.
- 2) Nachdenken über das Programm, Ziehen von Schlussfolgerungen (z.B. Verhaltenshypothesen).

Zur Überprüfung von Verhaltenshypothesen, aber auch zu deren Auffinden kann hilfreich sein:

- 3) Gedankliche und/oder 'Papier- und Bleistift'-Programmausführung.

Auf einer konzeptuell anderen Ebene kann das Verständnis des Ressourcenbedarfs helfen, ein Programm zu verstehen:

- 4) Analyse des Zeit- und Speicherplatzverhaltens eines Programms.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

735/170

# Ein Beispiel

...zur Illustration:

```
mapWhile :: (a -> b) -> (a -> Bool) -> [a] -> [b]
mapWhile f p [] = []                                (mW1)
mapWhile f p (x:xs)
  | p x          = f x : mapWhile f p xs              (mW2)
  | otherwise    = []                                (mW3)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

736/1700



# (1) Lesen des Programmtexts

Lesen der Funktionssignatur liefert bereits Einsichten in Art und Typ von Argumenten und Resultat. Im Beispiel:

- ▶ `mapWhile` erwartet als Argumente eine
  - ▶ Funktion `f` eines nicht weiter eingeschränkten Typs  $(a \rightarrow b)$
  - ▶ Eigenschaft `p` von Objekten vom Typ `a`, genauer ein Prädikat (oder Wahrheitswertfunktion) vom Typ  $(a \rightarrow \text{Bool})$
  - ▶ Liste `l` von Elementen vom Typ `a`

`mapWhile` liefert als Resultat

- ▶ eine Liste `l'` von Elementen vom Typ `b`

...Lesen zusätzlich eingestreuter Programmkommentare, möglicherweise auch in Form von Vor- und Nachbedingungen ermöglicht (hoffentlich)

- ▶ weitere und tiefergehende Einsichten.

# (1) Lesen des Programmtexts (fgs.)

Lesen der Funktionsdefinition liefert erste weitere Einsichten in Verhalten und Bedeutung des Programms. Im Beispiel:

- ▶ Angewendet auf die leere Liste `[]`, ist gemäß (mW1) das Resultat die leere Liste `[]`.
- ▶ Angewendet auf eine nichtleere Liste, deren Kopfelement `x` Eigenschaft `p` erfüllt, ist gemäß (mW2) das Element `(f x)` vom Typ `b` das Kopfelement der Resultatliste, deren Rest sich durch den rekursiven Aufruf auf die Restliste `xs` ergibt.
- ▶ Erfüllt Kopfelement `x` die Eigenschaft `p` nicht, bricht gemäß (mW3) die Berechnung ab und liefert als Resultat die leere Liste `[]` zurück.

## (2) Nachdenken über das Programm

**Nachdenken** liefert tiefere Einsichten über **Programmverhalten** und **-bedeutung**, auch durch den **Beweis von Eigenschaften**, die das Programm besitzt. Im Beispiel:

- Für alle Funktionen **f**, Prädikate **p** und endliche Listen **xs** können wir folgende Gleichheiten beweisen:

`mapWhile f p xs = map f (takeWhile p xs)` (mW4)

`mapWhile f (const True) xs = map f xs` (mW5)

`mapWhile id p xs = takeWhile p xs` (mW6)

wobei (mW5) und (mW6) Folgerungen aus (mW4) sind.

### (3) Gedankliche, Papier- u. Bleistiftausführung

...hilft, **Verhaltenshypothesen** zu **validieren** oder zu **generieren** durch Berechnung der Funktionswerte für ausgewählte Argumente. Im Beispiel:

```
mapWhile (+1) (>=7) [8,12,7,3,16]
->> (8+1) : mapWhile (+1) (>=7) [12,7,3,16]   wg. (mW2)
->> 9 : (12+1) : mapWhile (+1) (>=7) [7,3,16]   wg. (mW2)
->> 9 : 13 : (7+1) : mapWhile (+1) (>=7) [3,16]  wg. (mW2)
->> 9 : 13 : 8 : []                             wg. (mW3)
->> [9,13,8]
```

```
mapWhile (+1) (>=0) [8,12,7,3,16]
->> ...
->> [9,13,8,4,17]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

740/1700

## (4) Analyse des Ressourcenverbrauchs

...des Programms liefert:

- Für das **Zeitverhalten**: Unter der Annahme, dass **f** und **p** jeweils in konstanter Zeit ausgewertet werden können, ist die Auswertung von **mapWhile linear** in der Länge der Argumentliste, da im schlechtesten Fall die gesamte Liste durchgegangen wird.
- Für das **Speicherverhalten**: Der Platzbedarf ist **konstant**, da das Kopfelement stets schon 'ausgegeben' werden kann, sobald es berechnet ist (siehe unterstrichene Resultateile):

```
mapWhile (+1) (>=7) [8,12,7,3,16]
->> (8+1) : mapWhile (+1) (>=7) [12,7,3,16]
->> 9 : (12+1) : mapWhile (+1) (>=7) [7,3,16]
->> 9 : 13 : (7+1) : mapWhile (+1) (>=7) [3,16]
->> 9 : 13 : 8 : []
->> [9,13,8]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

# Zusammenfassung (1)

...jede der vorgestellten 4 Vorgangsweisen

- ▶ bietet einen anderen Zugang zum Verstehen eines Programms.
- ▶ liefert für sich einen Mosaikstein zu seinem Verstehen, aus denen sich durch Zusammensetzen ein vollständig(er)es Gesamtbild ergibt.
- ▶ kann 'von unten nach oben' auch auf Systeme von auf sich wechselseitig abstützenden Funktionen angewendet werden.
- ▶ bietet mit Vorgangsweise (3) der gedanklichen oder Papier- und Bleistiftausführung eines Programms einen stets anwendbaren (Erst-) Zugang zum Erschließen der Programmbedeutung an.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

# Zusammenfassung (2)

Lesbarkeit und Verständlichkeit eines Programms sollten

- ▶ immer schon beim Schreiben des Programms bedacht werden

...nicht zuletzt im eigenen Interesse!

Programme können grundsätzlich  
auf zwei Arten geschrieben werden:

So einfach, dass sie offensichtlich keinen Fehler enthalten;  
so kompliziert, dass sie keinen offensichtlichen Fehler enthalten.

C.A.R. 'Tony' Hoare (\* 1934)

*Turing Award* Preisträger 1980

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

# Kapitel 9.3

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

**9.3**

Teil IV





Kap. 10

Kap. 11

Teil V



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 9 (1)

-  Matthias Felleisen, Rober B. Findler, Matthew Flatt, Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.
-  Hugh Glaser, Pieter H. Hartel, Paul W. Garrat. *Programming by Numbers: A Programming Method for Novices*. The Computer Journal 43(4):252-265, 2000.
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Kapitel 6.6, Advice on Recursion)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 10, Functionally Solving Problems)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3




Teil IV

Kap. 10

Kap. 11

Teil V

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 9 (2)

-  Norman Ramsey. *On Teaching How to Design Programs*. In Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP 2014), 153-166, 2014.
-  Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik. Induktives Vorgehen*. Springer-V., 2014. (Kapitel 4, Induktives Definieren)
-  Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018. (Kapitel 4, Inductive Definitions)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 9 (3)



Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999.  
(Kapitel 7.4, Finding primitive recursive definitions; Kapitel 14, Designing and writing programs; Kapitel 11, Program development; Anhang D, Understanding programs)



Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.  
(Kapitel 4, Designing and writing programs; Kapitel 7.4, Finding primitive recursive definitions; Kapitel 9.1, Understanding definitions; Kapitel 12.7, Understanding programs)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

9.1

9.2

9.3

Teil IV

Kap. 10

Kap. 11

Teil V

# Teil IV

## Funktionale Programmierung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

**Teil IV**

Kap. 10

Kap. 11

Teil V

Kap. 12

748/170

# Kapitel 10

## Funktionen höherer Ordnung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

**Kap. 10**

10.1

10.2

10.3

10.4

10.5

10.6

# Kapitel 10.1

## Motivation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

**10.1**

10.1.1

10.1.2

10.1.3

10.1.4

10.2

750/170

# Funktionen höherer Ordnung

...Bezeichnung für **Funktionen**, unter deren **Argumenten** oder **Resultaten** Funktionen sind.

Damit gilt:

**Funktionen höherer Ordnung** (oder kurz **Funktionale**) sind

- **spezielle Funktionen**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.1.1

10.1.2

10.1.3

10.1.4

10.2

751/170

# Kapitel 10.1.1

## Beispiele vordefinierter Funktionale

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

**10.1.1**

10.1.2

10.1.3

10.1.4

10.2

752/170



# Beispiele vordefinierter Funktionale in Haskell

## Funktionen mit funktionalen Resultaten:

```
(+) :: Num a => a -> (a -> a)
((+) 1) :: Num a => (a -> a)           -- Inkrementfkt.
splitAt :: Int -> ([a] -> ([a],[a]))
(splitAt 42) :: ([a] -> ([a],[a]))    -- Listenteilungsfkt.
```

## Funktionen mit funktionalen Argumenten und Resultaten:

```
curry :: ((a,b) -> c) -> (a -> (b -> c))
(curry binom') :: (Integer -> (Integer -> Integer))
                                           -- binom-Fkt.

uncurry :: (a -> (b -> c)) -> ((a,b) -> c)
(uncurry binom) :: ((Integer,Integer) -> Integer)
                                           -- binom'-Fkt.

zipWith :: (a -> (b -> c)) -> ([a] -> ([b] -> [c]))
(zipWith (&&)) :: ([Bool] -> ([Bool] -> [Bool]))
                                           -- 'Elementweise-und'-Fkt.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.1.1

10.1.2

10.1.3

10.1.4

10.2

753/170

# Kapitel 10.1.2

## Beispiele selbstdefinierter Funktionale

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.1.1

**10.1.2**

10.1.3

10.1.4

10.2

754/170

# Beispiele selbstdef. Funktionale in Haskell

Funktionen mit funktionalen Argumenten (und nichtfunktionalen Resultaten):

```
f :: ((a -> b), a) -> b
```

```
f (g,x) = g x
```

```
f (fac,5) = 120 :: Integer
```

```
f (reverse,"stressed") = "desserts" :: String
```

```
f (concat,[[ 'a','b','c'],[ 'd','e'],[],[ 'f']])  
  = [ 'a','b','c','d','e','f'] :: [Char]
```

```
...
```

```
h :: ((a -> b -> c), a, b) -> c
```

```
h (g,x,y) = g x y
```

```
h (binom,49,6) = 13.983.816 :: Integer
```

```
h (++),"Hallo"," Welt!") = "Hallo Welt!" :: String
```

```
h (zip,[ 'a','b','c'],[True,False])  
  = ([ ('a',True), ('b',False)] :: [(Char,Bool)])
```

```
...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.1.1

10.1.2

10.1.3

10.1.4

10.2

755/170

# Bemerkung

...funktionale Programmiersprachen und Programmierung haben eine Präferenz für curryfizierte Funktionsdefinitionen.

Das erklärt die

- ▶ Abwesenheit vordefinierter Funktionale mit funktionalen Argumenten ohne funktionale Resultate

in Haskell.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.1.1

10.1.2

10.1.3

10.1.4

10.2

756/170

# Die Eingangsbeispiele

...zeigen:

Funktionen höherer Ordnung kommen in funktionalen Sprachen

- ▶ völlig **beiläufig** und **natürlich** daher.

So **beiläufig**, dass sie in **funktionaler Programmierung**

- ▶ der **Regelfall**, nicht die **Ausnahme** sind.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.1.1

**10.1.2**

10.1.3

10.1.4

10.2

757/170

# Kapitel 10.1.3

## Beispiele aus der Mathematik

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.1.1

10.1.2

**10.1.3**

10.1.4

10.2

758/170

# Funktionen höherer Ordnung

...in der Mathematik, z.B. in der Analysis:

## ► Differentialrechnung:

$\frac{df(x)}{dx} \rightsquigarrow$  ableitung  $f$   $x$   
...Steigung der Funktion  $f$  an der Stelle  $x$ .

## ► Integralrechnung:

$\int_a^b f(x) dx \rightsquigarrow$  integral  $f$   $a$   $b$   
...Fläche unterhalb d. Fkt.  $f$  zwischen  $a$  und  $b$ .

## ► Stetigkeitstheorem:

Die Komposition zweier stetiger Funktionen ist eine stetige Funktion, d.h. die Komposition der Funktionen  $f, g : \mathbb{R} \rightarrow \mathbb{R}$ ,  $(f \circ g) : (\mathbb{R} \rightarrow \mathbb{R}) \times (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$ , mit  $(f \circ g)(x) = f(g(x))$  ist stetig, wenn  $f$  und  $g$  stetig sind.

# Kapitel 10.1.4

## Beispiele aus anderen Informatikbereichen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.1.1

10.1.2

10.1.3

**10.1.4**

10.2

760/1700



# Funktionen höherer Ordnung

...in anderen Informatikbereichen, z.B. der Semantik von Programmiersprachen:

- ▶ Die Bedeutung der while-Schleife im denotationellen Stil

$$\llbracket \text{while } b \text{ do } \pi \text{ od} \rrbracket_{ds} : \Sigma \rightarrow \Sigma$$

...als kleinster Fixpunkt der Funktion höherer Ordnung *FIX* auf der Menge der Zustandstransformationen mit

- ▶  $V$ : Menge der Programmvariablen;  $D$ : Datenbereich.
- ▶  $\Sigma =_{df} \{ \sigma \mid \sigma : V \rightarrow D \}$ : Menge der (Prg.-) Zustände.
- ▶  $ZT = [\Sigma \rightarrow \Sigma] =_{df} \{ zt \mid zt : \Sigma \rightarrow \Sigma \}$   
 $= \{ zt \mid zt : (V \rightarrow D) \rightarrow (V \rightarrow D) \}$ :  
Menge der (Prg.-) Zustandstransformationen.
- ▶  $FIX : ((\Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma)) \rightarrow (\Sigma \rightarrow \Sigma)$ :  
Fixpunktzustandstransformationsfunktional, das den  
kleinsten Fixpunkt des Argumentfunktionals liefert.

(siehe z.B. VU 185.278 Theoretische Informatik und Logik)

# Die aufgefaltete Signatur

...des Fixpunktzustandstransformationsfunktionals:

$$\text{FIX} : (((V \rightarrow D) \rightarrow (V \rightarrow D)) \rightarrow ((V \rightarrow D) \rightarrow (V \rightarrow D))) \\ \rightarrow ((V \rightarrow D) \rightarrow (V \rightarrow D))$$

$$\text{FIX} : (((\underbrace{(V \rightarrow D)}_{\text{Zustandsfkt.}} \rightarrow \underbrace{(V \rightarrow D)}_{\text{Zustandsfkt.}})) \rightarrow ((\underbrace{(V \rightarrow D)}_{\text{Zustandsfkt.}} \rightarrow \underbrace{(V \rightarrow D)}_{\text{Zustandsfkt.}}))) \\ \underbrace{\hspace{10em}}_{\text{Zustandstransf.-fkt.}} \hspace{10em} \underbrace{\hspace{10em}}_{\text{Zustandstransf.-fkt.}} \\ \underbrace{\hspace{15em}}_{\text{Zustandstransformationsfunktionenfunktion}} \\ \rightarrow ((\underbrace{(V \rightarrow D)}_{\text{Zustandsfkt.}} \rightarrow \underbrace{(V \rightarrow D)}_{\text{Zustandsfkt.}})) \\ \underbrace{\hspace{10em}}_{\text{Zustandstransformationsfunktion}}$$

Zustandsfunktionentransformationsfunktionenfunktionenfixpunktfunktion

# Funktionen erster und höherer Ordnung

...am Beispiel von *FIX*:

- Zustandsfunktion:

$$(V \rightarrow D)$$

Funktion 1. Ordnung: Elementare Werte werden auf elementare Werte abgebildet.

- Zustandsfunktionstransformationsfunktion:

$$(V \rightarrow D) \rightarrow (V \rightarrow D)$$

Funktion höherer Ordnung 1. Stufe: Funktionen 1. Ordnung werden auf Funktionen 1. Ordnung abgebildet.

- Zustandsfunktionstransformationsfunktionenfunktion:

$$((V \rightarrow D) \rightarrow (V \rightarrow D)) \rightarrow ((V \rightarrow D) \rightarrow (V \rightarrow D))$$

Funktion höherer Ordnung 2. Stufe: Funktionen höherer Ordnung 1. Stufe werden auf Funktionen höherer Ordnung 1. Stufe abgebildet.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.1.1

10.1.2

10.1.3

10.1.4

10.2

763/170

# Funktionen erster und höherer Ordnung (fgs.)

- Zustandsfunktionen transformationsfunktionen funktionenfixpunktfunktion:

$$\begin{aligned} &(((V \rightarrow D) \rightarrow (V \rightarrow D)) \rightarrow ((V \rightarrow D) \rightarrow (V \rightarrow D))) \\ &\quad \rightarrow ((V \rightarrow D) \rightarrow (V \rightarrow D)) \end{aligned}$$

Funktion höherer Ordnung 3. Stufe: Funktionen höherer Ordnung 2. Stufe werden auf Funktionen höherer Ordnung 1. Stufe abgebildet.

...oder kürzer:

Fixpunktzustandstransformationsfunktional!

# Funktionen höherer Ordnung

...können dennoch überraschen oder gar verstören:

*“The functions I grew up with, such as the sine, the cosine, the square root, and the logarithm were almost exclusively real functions of a real argument.*

*[...] I was really ill-equipped to appreciate functional programming when I encountered it: I was, for instance, totally baffled by the shocking suggestion that the value of a function could be another function.”<sup>(\*)</sup>*

Edsger W. Dijkstra (1930-2002)  
*Turing Award* Preisträger 1972

<sup>(\*)</sup> Zitat aus: Introducing a course on calculi. Ankündigung einer Lehrveranstaltung an der University of Texas, Austin, 1995.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.1.1

10.1.2

10.1.3

10.1.4

10.2

765/170

# Mit Funktionen höherer Ordnung

...machen wir den Schritt von **applikativer** zu **funktionaler Programmierung**!

Frei nach **Hegel**:

Der **Mensch**  
wird erst durch **Arbeit**  
zum **Menschen**.

Georg W.F. Hegel (1770-1831)  
dt. Philosoph

...auf den **Punkt** gebracht:

*Die funktionale Programmierung  
wird erst durch Funktionen höherer Ordnung  
zu funktionaler Programmierung.*

# Mit Fug und Recht

...die vollumfängliche Integration von **Funktionen höherer Ordnung** als **erstrangige Elemente** (engl. **first-class citizens**)

- ▶ ist charakteristisch und kennzeichnend für **funktionale Programmierung**.
- ▶ hebt **funktionale Programmierung** von anderen Programmierparadigmen ab.
- ▶ ist wesentliches sprachliches Mittel **funktionaler Sprachen** für extrem ausdruckskräftige, elegante und flexible Programmiermethoden, insbesondere zur Unterstützung von **Wiederverwendung**.

# Kapitel 10.2

## Funktionale Abstraktion

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

**10.2**

10.2.1

10.2.2

10.2.3

10.3

768/170



# Abstraktionsprinzipien

Kennzeichnendes **Strukturierungsprinzip** für

- ▶ **Prozedurale Sprachen**: **Prozedurale Abstraktion**
  - ▶ Operanden werden zu Parametern von **Prozeduren**.
- ▶ **Funktionale Sprachen**: **Funktionale Abstraktion**
  - ▶ 1-ter Stufe: **Funktionen**
    - ↪ Nichtfunktionale Operanden werden zu Parametern von **Funktionen** (funktionales Analogon zu **prozeduraler Abstraktion**).
  - ▶ Höherer Stufe: **Funktionen höherer Ordnung**
    - ↪ Verknüpfungsvorschriften werden zu funktionalen Parametern von **Funktionen höherer Ordnung**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

10.2.3

10.3

769/170

# Kapitel 10.2.1

## Funktionale Abstraktion 1. Stufe

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

**10.2.1**

10.2.2

10.2.3

10.3

770/170

# Funktionale Abstraktion 1-ter Stufe (1)

Idee: Operanden werden zu Parametern von Funktionen.

Beispiel: Statt viele strukturell gleiche Ausdrücke wie

$(5 * 37 + 13) * (37 + 5 * 13)$

$(15 * 7 + 12) * (7 + 15 * 12)$

$(25 * 3 + 10) * (3 + 25 * 10)$

...

...immer wieder von vorn hinschreiben und auszuwerten, führe eine funktionale Abstraktion durch, d.h. schreibe eine Funktion, die die Operanden des Ausdrucksmusters als Parameter erhält:

$f :: (\text{Int}, \text{Int}, \text{Int}) \rightarrow \text{Int}$

$f(a, b, c) = (a * b + c) * (b + a * c)$

und mit den ursprünglichen Ausdrucksooperanden(werten) aufgerufen wird.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

10.2.3

10.3

771/170

# Funktionale Abstraktion 1-ter Stufe (2)

Beispiel (fgs.): Die Funktion `f` erlaubt uns die gemeinsame Berechnungsvorschrift  $(a * b + c) * (b + a * c)$  der strukturell gleichen Ausdrücke **wiederverwenden**:

`f (5,37,13) ->> 20.196`

`f (15,7,12) ->> 21.879`

`f (25,3,10) ->> 21.930`

...

Gewinn: Wiederverwendung der gemeinsamen Berechnungsvorschrift durch

► funktionale Abstraktion.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

10.2.3

10.3

772/1700

# Kapitel 10.2.1

## Funktionale Abstraktion höherer Stufe

# Funktionale Abstraktion höherer Stufe (1)

Idee: Verknüpfungsvorschriften werden zu funktionalen Parametern einer Funktion, die Funktion dadurch zu einer Funktion höherer Ordnung.

Beispiel: (siehe Fethi Rabhi, Guy Lapalme. *Algorithms - A Functional Approach*, Addison-Wesley, 1999, S. 7f.):

► Fakultätsfunktion:

$$\begin{array}{lcl} \text{fac } n & | & n==0 \\ & | & n>0 \end{array} \quad \begin{array}{l} = 1 \\ = n * \text{fac } (n-1) \end{array}$$

► Summe der  $n$  ersten natürlichen Zahlen:

$$\begin{array}{lcl} \text{natSum } n & | & n==0 \\ & | & n>0 \end{array} \quad \begin{array}{l} = 0 \\ = n + \text{natSum } (n-1) \end{array}$$

► Summe der  $n$  ersten natürlichen Quadratzahlen:

$$\begin{array}{lcl} \text{natQuSum } n & | & n==0 \\ & | & n>0 \end{array} \quad \begin{array}{l} = 0 \\ = n*n + \text{natQuSum } (n-1) \end{array}$$

# Funktionale Abstraktion höherer Stufe (2)

## Beobachtung:

- ▶ Die Definitionen von `fac`, `natSum` und `natQuSum` folgen demselben **Rekursionsschema** und der strukturell selben **Verknüpfungsvorschrift** ihrer Argumente.

Dieses gemeinsame **Rekursionsschema** und die **Verknüpfungsvorschrift** sind gekennzeichnet durch die Festlegung von im

- ▶ **Basisfall**: eines **Basiswerts**.
- ▶ **Rekursionsfall**: einer **Verknüpfungsvorschrift** des Argumentwerts `n` und des Funktionswerts für `(n-1)`.

# Funktionale Abstraktion höherer Stufe (3)

Diese **Gemeinsamkeit** legt es nahe

- ▶ **Rekursionsschema**
- ▶ **Verknüpfungsvorschrift**
- ▶ **Basiswert**

herauszuziehen, zu **abstrahieren**; eine Abstraktion höherer Stufe.

Das ergibt folgendes **Rekursionsschema**:

```
rekSchema :: Int -> (Int -> Int -> Int) -> Int -> Int
rekSchema basiswert verknuepfe n
  | n==0 = basiswert
  | n>0  = verknuepfe n (rekSchema basiswert verknuepfe (n-1))
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

10.2.3

10.3

776/170



# Funktionale Abstraktion höherer Stufe (4)

...diese funktionale Abstraktion höherer Stufe erlaubt nun, die Implementierungen von

► `fac`, `natSum` und `natQuSum`

zu ersetzen durch passende Aufrufe der

► Funktion höherer Ordnung `rekSchema`

der die Verknüpfungsvorschriften von `fac`, `natSum` und `natQuSum` über den funktionalen Parameter `verknuepfe` übergeben werden.

# Funktionale Abstraktion höherer Stufe (5)

Redefinition der Funktionen mittels `rekSchema`:

`fac` = `rekSchema 1 (*)`

`natSum` = `rekSchema 0 (+)`

`natQuSum` = `rekSchema 0 (\x y -> x*x + y)`

...alternativ `argumentbehaftet`:

`fac n` = `rekSchema 1 (*) n`

`natSum n` = `rekSchema 0 (+) n`

`natQuSum n` = `rekSchema 0 (\x y -> x*x + y) n`

**Gewinn:** Wiederverwendung des gemeinsamen Strukturmusters der Funktionen `fac`, `natSum` und `natQuSum` durch

► **funktionale Abstraktion höherer Stufe.**

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

10.2.3

10.3

778/170

# Zusammenfassung d. rekSchema-Beispiels (1)

...die Signatur zeigt, dass **rekSchema** eine **Funktion höherer Ordnung** ist, die als ein Argument eine **Funktion** erwartet:

**rekSchema** :: Int -> (Int -> Int -> Int) -> Int -> Int

**Beachte:** Streng genommen, ist **rekSchema** eine einstellige Funktion, die aufgerufen mit einem ganzzahligen Argument **z** eine **Funktion höherer Ordnung** als Resultat liefert, nämlich den Wert des Funktionsterms **(rekSchema z)** vom Typ

**(rekSchema z)** :: (Int -> Int -> Int) -> Int -> Int

Die uncurryfizierte Version von **rekSchema** bzw. mit getauschter Argumentfolge macht deutlicher, dass das Rekursionsschema (u.a.) eine Funktion als Argument erwartet:

**rekSchema'** :: (Int, (Int -> Int -> Int), Int) -> Int

**rekSchema''** :: (Int -> Int -> Int) -> Int -> Int -> Int

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

10.2.3

10.3

779/170

# Zusammenfassung d. rekSchema-Beispiels (2)

Für die Anwendungsbeispiele von **rekSchema** gilt:

	Basiswert	Verknüpfungsvorschrift
fac	1	(*)
natSum	0	(+)
natQuSum	0	$\backslash x \ y \rightarrow x*x + y$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

10.2.3

10.3

780/170

# Übungsaufgabe 10.2.2.1

Ergänze die Deklarationen von

`rekSchema'` :: (Int, (Int -> Int -> Int), Int) -> Int

`rekSchema''` :: (Int -> Int -> Int) -> Int -> Int -> Int

zu vollständigen Implementierungen und teste sie mit geeigneten Argumenten.

# Zurück zum u. weiter mit d. Eingangsbsp. (1)

- Funktionale Abstraktion 1. Stufe führt von Ausdrücken  
 $(5*37+13)*(37+5*13)$ ,  $(15*7+12)*(7+15*12)$ , ...  
zu Funktionen:

$f :: (\text{Int}, \text{Int}, \text{Int}) \rightarrow \text{Int}$

$f(a, b, c) = (a * b + c) * (b + a * c)$

Aufrufbeispiele:

$f(5, 37, 13) \rightarrow 20.196$

$f(15, 7, 12) \rightarrow 21.879$

...

- Funktionale Abstraktion höherer Stufe führt von Funktionen zu Funktionen höherer Ordnung:

$fho :: ((\text{Int}, \text{Int}, \text{Int}) \rightarrow \text{Int}), \text{Int}, \text{Int}, \text{Int}) \rightarrow \text{Int}$

$fho(g, a, b, c) = g(a, b, c)$

Aufrufbeispiele:

$fho(f, 5, 37, 13) \rightarrow 20.196$

$fho(f, 15, 7, 12) \rightarrow 21.879$

...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

10.2.3

10.3

782/170

# Zurück zum u. weiter mit d. Eingangsbsp. (2)

...zusätzlich zur

► **freien Wahl** der elementaren Argumentwerte

(wie **f**) erlaubt die **Funktion höherer Ordnung fho** auch die

► **freie Wahl** der Vorschrift sie zu **verknüpfen**.

Beispiele:

```
f :: Int -> Int -> Int -> Int
```

```
f a b c = (a * b + c) * (b + a * c)
```

```
g :: Int -> Int -> Int -> Int
```

```
g a b c = a^b 'div' c
```

```
h :: Int -> Int -> Int -> Int
```

```
h a b c = if (a 'mod' 2 == 0) then b else c
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

10.2.3

10.3

783/1700

# Zurück zum u. weiter mit d. Eingangsbsp. (3)

## Aufrufbeispiele:

```
fho (f,2,3,5) ->> f 2 3 5
                ->> (2*3+5)*(3+2*5)
                ->> (6+5)*(3+10)
                ->> 11*13
                ->> 143
```

```
fho (g,2,3,5) ->> g 2 3 5
                ->> 2^3 'div' 5
                ->> 8 'div' 5
                ->> 1
```

```
fho (h,2,3,5) ->> h 2 3 5
                ->> if (2 'mod' 2 == 0) then 3 else 5
                ->> if (0 == 0) then 3 else 5
                ->> if True then 3 else 5
                ->> 3
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

10.2.3

10.3

784/170



# Kapitel 10.2.3

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.2.1

10.2.2

**10.2.3**

10.3

# Zusammenfassung

...Gewinn durch funktionale Abstraktion:

- Wiederverwendung

und dadurch kürzerer, verlässlicherer, wartungsfreundlicherer Code.

Zwingend erforderlich für erfolgreiches Gelingen:

- Funktionen höherer Ordnung (oder kurz Funktionale).

Als Abschluss:

- Der allgemeinste Typ des Funktional `rekSchema` (s.a. Kap. 11 und Kap. 14) ist:

```
rekSchema :: (Num a, Ord a) =>  
             b -> (a -> b -> b) -> a -> b
```

# Kapitel 10.3

## Funktionen als Argument

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

**10.3**

10.3.1

10.3.2

10.4

# Kapitel 10.3.1

## Beispiele

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

**10.3.1**

10.3.2

10.4

788/170

# Funktionen als Argument: 1-tes Beispiel (1)

Betrachte die spezialisierten Vergleichsfunktionen `min`, `max`:

```
min :: Ord a => a -> a -> a
```

```
min x y
```

```
  | x < y      = x
```

```
  | otherwise = y
```

```
max :: Ord a => a -> a -> a
```

```
max x y
```

```
  | x > y      = x
```

```
  | otherwise = y
```

...Abstraktion höherer Stufe und herausziehen der Vergleichsoperation erlaubt die Vergleichsfunktionen zu generalisieren...

# Funktionen als Argument: 1-tes Beispiel (2)

...zu einer mit einer **Wahrheitswertfunktion** parametrisierten Funktion höherer Ordnung **extreme**, die **min**, **max** zu redefinieren erlaubt:

```
extreme :: Ord a => (a -> a -> Bool) -> a -> a -> a
extreme wwf m n
  | wwf m n    = m
  | otherwise  = n

min = extreme (<)           -- argumentfrei
max = extreme (>)

min x y = extreme (<) x y   -- argumentbehaftet
max x y = extreme (>) x y
```

...oder auch gänzlich (durch Aufrufe v. **extreme**) zu ersetzen:

```
max 17 4 ->> extreme (>) 17 4 ->> 17
min 17 4 ->> extreme (<) 17 4 ->> 4
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.4

790/1700

# Funktionen als Argument: 2-tes Beispiel (1)

Betrachte die Funktion `zip`:

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []
```

...und die Funktion höherer Ordnung `zipWith`:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = []
```

`zipWith` erlaubt `zip` zu implementieren (und zu ersetzen):

```
zip :: [a] -> [b] -> [(a,b)]
zip xs ys = zipWith v xs ys
              where v :: a -> b -> (a,b)
                    v = (,) -- (,) Paarbildungsop.

-- v x y = (x,y) gleichbedeutend zu: v x y = (,) x y
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.4

791/1700

# Funktionen als Argument: 2-tes Beispiel (2)

...aufgrund der Parametrisierung leistet `zipWith` mehr als `zip` zu implementieren (und ist in diesem Sinn genereller).

Betrachte dazu etwa folgende Beispiele:

```
f :: a -> b -> (a,b)
```

```
f x y = (x,y)
```

```
g :: a -> a -> [a]
```

```
g x y = [x,y]
```

```
h :: Num a => a -> a -> a
```

```
h x y = x+y
```

```
k :: Ord a => a -> a -> Bool
```

```
k x y = x > y
```

```
zipWith f ['a','b'] [1,2,3] ->> [('a',1),('b',2)]
```

```
zipWith g [1,2,3] [5,6,7,8] ->> [[1,5],[2,6],[3,7]]
```

```
zipWith h [1,2,3] [10,20,30,40] ->> [11,22,33]
```

```
zipWith k [10,20,30] [5,15,35,85] ->> [True,True,False]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.4

792/1700



# Funktionen als Argument: 3-tes Beispiel

**Transformation** der Marken eines benannten Baums bzw. **Herausfiltern** der Marken mit einer bestimmten Eigenschaft:

```
data Baum a = Leer | Wurzel a (Baum a) (Baum a)

map_Baum :: (a -> a) -> Baum a -> Baum a
map_Baum _ Leer = Leer
map_Baum tf (Wurzel marke ltb rtb) =
  Wurzel (tf marke) (map_Baum tf ltb) (map_Baum tf rtb)

filter_Baum :: (a -> Bool) -> Baum a -> [a]
filter_Baum _ Leer = []
filter_Baum ww (Wurzel marke ltb rtb)
  | ww marke = marke : ((filter_Baum ww ltb)
                        ++ (filter_Baum ww rtb))
  | otherwise = (filter_Baum ww ltb)
                ++ (filter_Baum ww rtb)
```

...mithilfe zweier Funktionen höherer Ordnung, die parametrisiert sind in **Transformationsfunktion** bzw. **Wahrheitswertfunktion**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.4

793/1700

# Kapitel 10.3.2

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.3.1

**10.3.2**

10.4

794/170

# Zusammenfassung

## Funktionen als Argument

- ▶ erhöhen die **Ausdruckskraft**.
- ▶ unterstützen **Wiederverwendung**.
- ▶ sind charakteristisch für **funktionale Programmierung**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.3.1

**10.3.2**

10.4

795/1700

# Kapitel 10.4

## Funktionen als Resultat

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

**10.4**

10.4.1

10.4.2

# Kapitel 10.4.1

## Beispiele

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

**10.4.1**

10.4.2

# Funktionen als Resultat (1)

...der **Regelfall**, nicht die Ausnahme in **funktionalen Sprachen**.

Betrachte zum **Beispiel**:

$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

$\text{binom} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$

$\text{rekSchema} :: (\text{Num } a, \text{Ord } a) \Rightarrow$   
 $\quad b \rightarrow (a \rightarrow b \rightarrow b) \rightarrow a \rightarrow b$

...

**Klammerung** hebt die **funktionalen Resultate** besonders hervor:

$(+) :: \text{Num } a \Rightarrow a \rightarrow (a \rightarrow a)$

$\text{binom} :: \text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer})$

$\text{rekSchema} :: (\text{Num } a, \text{Ord } a) \Rightarrow$   
 $\quad b \rightarrow ((a \rightarrow b \rightarrow b) \rightarrow (a \rightarrow b))$

...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

798/170

# Funktionen als Resultat (2)

## Wiederholtes Anwenden:

```
iterate :: Int -> (a -> a) -> (a -> a)
iterate n f
  | n > 0      = f . iterate (n-1) f  -- (.) Funktions-
                                       -- komposition
  | otherwise = id
where
  id :: a -> a          -- Typvariable und Parameter
  id a = a             -- dürfen gleichbenannt sein.

(iterate 3 square) 2
->> (square . square . square . id) 2 ->> 256
```

## Vertauschen von Argumenten:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x

flip (-) 3 5 ->> (-) 5 3 ->> 2
(flip . flip) ->> id
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

799/1700

# Kapitel 10.4.2

## Methoden 1 bis 6

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

**10.4.2**

800/1700



# Funktionen als Resultat: Methode 1 (1)

...explizites Ausprogrammieren:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
uncurry :: (a -> b -> c) -> ((a,b) -> c)
flip :: (a -> b -> c) -> (b -> a -> c)
iterate :: Int -> (a -> a) -> (a -> a)
extreme :: Ord a => (a -> a -> Bool) -> (a -> a -> a)
addFuns :: Num a => (a -> a) -> (a -> a) -> (a -> a)
addFuns f g = \x -> f x + g x
funny :: (Ord a, Num a) => (a -> a) -> (a -> a)
                                     -> (a -> a)
funny f g = \x -> if x >= 0 then (g . f) x
                      else addFuns f g (x+1)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

801/170

# Funktionen als Resultat: Methode 1 (2)

Vergleiche die Definitionen von `addFuns` und `addFuns'`:

`addFuns :: Num a => (a -> a) -> (a -> a) -> (a -> a)`

`addFuns f g = \x -> f x + g x`

`addFuns' :: Num a => (a -> a) -> (a -> a) -> (a -> a)`

`addFuns' f g x = f x + g x`

...und die `Typen` der zugehörigen definierenden Gleichungen:

`addFuns f g = \x -> f x + g x :: a -> a` (funktional)

`addFuns' f g x = f x + g x :: a` (nicht-funktional)

# Funktionen als Resultat: Methode 2

...partielle Auswertung curryfizierter Funktionen:

```
((+) 1) :: Num a => a -> a
(binom 45) :: Integer -> Integer
(rekSchema 0) :: (Num a, Ord a, Num b) =>
                  (a -> b -> b) -> (a -> b)
(rekSchema 0 (+)) :: (Num a, Ord a) => a -> b
(extreme (<)) :: Ord a => a -> a -> a
(extreme (<) 5) :: (Num a, Ord a) => a -> a
(iterate 5) :: (a -> a) -> (a -> a)
(iterate 5 fac) :: Integer -> Integer
(flip (-)) :: Num a => a -> a -> a
addFuns fac fac :: Integer -> Integer
funny fac fac :: Integer -> Integer
...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

803/170

# Funktionen als Resultat: Methode 3

...Operatorabschnitte (als Spezialfall partieller Auswertung für binäre Operatoren und Funktionen):

```
(+1) :: Num a => a -> a           -- Inkrementieren
(1-) :: Num a => a -> a           -- Eins_minus
(+(-1)) :: Num a => a -> a       -- Dekrementieren
(2*) :: Num a => a -> a           -- Verdoppeln
(<2) :: (Num a, Ord a) => a -> a  -- Kleiner 2?
(== True) :: Bool -> Bool        -- Wahr?
(True &&) :: Bool -> Bool        -- Wahr?
(42:) :: Num a => a -> [a] -> [a]
                                -- 42 als neuer Listenkopf
(45 'binom') :: Integer -> Integer -- 45 über k
(47 '(extreme (<))) :: (Num a, Ord a) => a -> a
                                -- Minimum aus 47 und x
```

...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

804/170

# Funktionen als Resultat: Methode 4

...Bildung konstanter Funktionen (engl.  $\lambda$ -Lifting):

```
lifting :: a -> (b -> a)
lifting a_const = \x -> a_const
```

Anwendungsbeispiele:

```
lifting 42 "Aller Fragen Antwort"      ->> 42
lifting iterate flip                    ->> iterate
lifting (iterate (+) 3 (\x->x*x)) 42 2 ->> 256
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

805/1700

# Funktionen als Resultat: Methode 5

...als **Abänderungen** gegebener Funktionen:

```
fac :: Integer -> Integer           -- argumentbehaftet
fac 0 = 1
fac n = n * fac (n-1)

fac' :: Integer -> Integer           -- argumentfrei
fac' =
  \n -> if n >= 0
        then fac n                -- Verhalten wie fac
        else (-1)                 -- Abweich. Verh. zu fac

fac'' :: Integer -> Integer          -- argumentfrei
fac'' =
  \n -> if n >= 0
        then fac n                -- Verhalten wie fac
        else fac (abs n)          -- Abw. Verh. zu fac
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

806/170

# Funktionen als Resultat: Methode 6

...Komposition von Funktionen:

$$(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

$$(f \cdot g) x = f (g x)$$

Wichtige Eigenschaft von  $(\cdot)$ : Assoziativität

$$(f \cdot (g \cdot h)) = ((f \cdot g) \cdot h) = (f \cdot g \cdot h)$$

Beachte: Funktionskomposition und Funktionsapplikation sind grundverschieden und auseinanderzuhalten:

► Komposition:  $(f \cdot g) x = f (g x) = f(g(x))$

► Applikation:  $(f g) x = (f g) x = (f(g))(x)$

# Übungsaufgabe 10.4.2.1

Überprüfe und teste die unterschiedliche Wirkung von Komposition und Applikation

► **Komposition:**  $(f \circ g) x = f (g x) = f(g(x))$

► **Applikation:**  $(f g) x = (f g) x = (f(g))(x)$

anhand geeigneter Beispiele für  $f$ ,  $g$  und  $x$ .

Beachte, dass sich eine Funktion  $f$ , die sich mit einer Funktion  $g$  komponieren lässt, nicht notwendig auf  $g$  applizieren lässt und umgekehrt.

Gibt es Beispiele für  $f$ ,  $g$  und  $x$ , so dass sowohl

$$(f \circ g) x$$

als auch

$$(f g) x$$

gültige Ausdrücke sind?



# Funktionskomposition: Anwendungsbsp. (1)

Das 4-te Element einer Liste:

```
gib_4tes_Element :: [a] -> a
gib_4tes_Element = head . dreimal_rest

dreimal_rest :: [a] -> [a]
dreimal_rest = tail . tail . tail
```

Das n-te Element einer Liste:

```
gib_ntes_Element :: Int -> [a] -> a
gib_ntes_Element n = (head . (iterate (n-1) tail))
```

...**Funktionskomposition** ermöglicht Funktionsdefinitionen auf dem (Abstraktions-) Niveau von Funktionen statt von (elementaren) Werten.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

809/170

# Funktionskomposition: Anwendungsbsp. (2)

...Definitionen auf Funktionsniveau sind **kürzer** und meist **einfacher zu verstehen** als ihre argumentbehafteten Gegenstücke.

Zum **Vergleich** einige **argumentfreie** und **argumentbehaftete** Implementierungen:

```
gib_4tes_Element :: [a] -> a
gib_4tes_Element = head . dreimal_rest

gib_4tes_Element ls = (head . dreimal_rest) ls
gib_4tes_Element ls = head (dreimal_rest ls)

gib_ntes_Element :: Int -> [a] -> a
gib_ntes_Element = head . (iterate tail)

gib_ntes_Element n = (head . (iterate tail)) n
gib_ntes_Element n lst
  = (head . (iterate tail) n) lst
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

810/170

# Kapitel 10.4.3

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

811/170

# Zusammenfassung

## Funktionen als Resultat

- ▶ erhöhen die **Ausdruckskraft**.
- ▶ unterstützen **Wiederverwendung**.
- ▶ sind kennzeichnend für **funktionale Programmierung**.

**Insgesamt:** Funktionen **gleichberechtigt** zu elementaren Werten als **Argument** und **Resultat** von Funktionen zuzulassen

- ▶ ist maßgeblich für **Ausdruckskraft**, **Eleganz** und **Prägnanz** funktionaler Programmierung.
- ▶ zeichnet **funktionale Programmierung** signifikant vor anderen Programmierparadigmen aus.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.4.1

10.4.2

812/170

# Kapitel 10.5

## Vordefinierte Funktionale auf Listen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

**10.5**

10.5.1

813/170

# Funktionale auf Listen

...ein wichtiger Spezialfall.

Vordefinierte **Listenfunktionale** für häufige Problemstellungen in **Haskell** (und anderen funktionalen Programmiersprachen):

- **Transformieren** aller Listenelemente mittels einer **Abbildungsvorschrift**:

```
map :: (a -> b) -> [a] -> [b]
```

- **Herausfiltern** aller Listenelemente mit einer bestimmten **Eigenschaft**:

```
filter :: (a -> Bool) -> [a] -> [a]
```

- **Aggregieren** aller Listenelemente mittels einer **Verknüpfungsoperation**:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

- ...

# Kapitel 10.5.1

## Transformieren: Das Funktional map

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

815/170

# Transformieren: Das Funktional map (1)

Signatur:

```
map :: (a -> b) -> [a] -> [b]
```

Implementierung mittels (expliziter) Rekursion:

```
map f []      = []  
map f (1:ls) = (f 1) : map f ls
```

Implementierung mittels Listenkompensation:

```
map f ls = [f l | l <- ls]
```

Anwendungsbeispiele:

```
map square [2,4..10] ->> [4,16,36,64,100]  
map length ["abc","abcde","ab"] ->> [3,5,2]  
map (>0) [4,(-3),2,(-1),0,2]  
->> [True,False,True,False,False,True]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

816/1700



# Transformieren: Das Funktional map (2)

## Anwendungsbeispiele (fgs.):

```
map (*) [2,4..10]
->> [(2*), (4*), (6*), (8*), (10*)] :: [Int -> Int]
map (-) [2,4..10]
->> [(2-), (4-), (6-), (8-), (10-)] :: [Int -> Int]
map (>) [2,4..10]
->> [(2>), (4>), (6>), (8>), (10>)] :: [Int -> Bool]

[f 10 | f <- map (*) [2,4..10] ]
->> [20,40,60,80,100]
[f 100 | f <- map (-) [2,4..10] ]
->> [-98,-96,-94,-92,-90]
[f 5 | f <- map (>) [2,4..10] ]
->> [False,False,True,True,True]
```

# Transformieren: Das Funktional map (3)

Einige **Eigenschaften** von **map**:

- Für **alle** Abbildungsvorschriften **f**, **g** gilt:

`map (\x -> x) = \x -> x`

`map (f . g) = map f . map g`

`map f . tail = tail . map f`

`map f . reverse = reverse . map f`

`map f . concat = concat . map (map f)`

`map f (xs ++ ys) = map f xs ++ map f ys`

- Für **strikte** (s. **Def. 13.4.1**) Abbildungsvorschriften **f** gilt:

`f . head = head . (map f)`

# Kapitel 10.5.2

## Filtern: Das Funktional `filter`

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

# Filtern: Das Funktional filter

Signatur:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Implementierung mittels (expliziter) Rekursion:

```
filter p []      = []  
filter p (l:ls)  = l : filter p ls  
                  | p l  
                  | otherwise = filter p ls
```

Implementierung mittels Listenkomprehension:

```
filter p ls = [l | l <- ls, p l]
```

Anwendungsbeispiel:

```
filter istZweierPotenz [2,4..100] ->> [2,4,8,16,32,64]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

820/170

## Kapitel 10.5.3

Aggregieren: Die Funktionale `foldl`, `foldr`

# Aggregieren, Falten von Listen: Motivation

**Aufgabe:** Berechne die Summe der Elemente einer Liste:

```
sum [1,2,3,4,5] ->> 15
```

Zwei Rechenweisen sind naheliegend zur Aufgabenlösung:

- **Summieren** (bzw. aggregieren, falten) **von rechts:**

```
(1+(2+(3+(4+5)))) ->> (1+(2+(3+9)))
```

```
->> (1+(2+12))
```

```
->> (1+14) ->> 15
```

- **Summieren** (bzw. aggregieren, falten) **von links:**

```
((((1+2)+3)+4)+5) ->> (((3+3)+4)+5)
```

```
->> ((6+4)+5)
```

```
->> (10+5) ->> 15
```

...die Funktionale **foldr** und **foldl** systematisieren diese Rechenweisen.

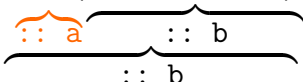
# Aggregieren: Das Funktional foldr (1)

Signatur (`foldr`: falten, zusammenfassen von rechts):

`foldr :: (a -> b -> b) -> b -> [a] -> b`

Implementierung mittels (expliziter) Rekursion:

`foldr f e [] = e`  
`foldr f e (l:ls) = f l (foldr f e ls)`



Es bedeuten:

- ▶ `f`: Faltungsvorschrift.
- ▶ `e`: Auffangwert, Vorgabewert für leere Argumentliste.
- ▶ `[]`, `(l:ls)`: Liste zu aggregierender Werte.

# Aggregieren: Das Funktional foldr (2)

## Anwendungsbeispiele:

```
foldr (+) 0 [] ->> 0
```

```
foldr (+) 0 [2,4..10]
```

```
->> ((+) 2 ((+) 4 ((+) 6 ((+) 8 ((+) 10 0))))
```

```
->> (2 + (4 + (6 + (8 + (10 + 0))))) ->> 30
```

```
foldr (*) 1 [] ->> 1
```

```
foldr (*) 1 [2,4..10]
```

```
->> ((*) 2 ((*) 4 ((*) 6 ((*) 8 ((*) 10 1))))
```

```
->> (2 * (4 * (6 * (8 * (10 * 1))))) ->> 3.840
```

```
foldr (||) False [] ->> False
```

```
foldr (||) False [True,False,False]
```

```
->> ((||) True ((||) False ((||) False False)))
```

```
->> (True || (False || (False || False))) ->> True
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

824/1700



# Aggregieren: Das Funktional foldr (3)

Anwendungsbeispiele (fgs.): Definition einiger Standardfunktionen in Haskell mittels `foldr`:

```
sum :: Num a => [a] -> a
```

```
sum ns = foldr (+) 0 ns
```

```
prod :: Num a => [a] -> a
```

```
prod ns = foldr (*) 1 ns
```

```
and :: [Bool] -> Bool
```

```
and bs = foldr (&&) True bs
```

```
or :: [Bool] -> Bool
```

```
or bs = foldr (||) False bs
```

```
concat :: [[a]] -> [a]
```

```
concat xss = foldr (++) [] xss
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

825/170

# Aggregieren: Das Funktional foldl (1)

Signatur (foldl: falten, zusammenfassen von links):

$\text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$

Implementierung mittels (expliziter) Rekursion:

$$\begin{aligned} \text{foldl } f \ e \ [] &= e \\ \text{foldl } f \ e \ (l:ls) &= \text{foldl } f \ (\underbrace{(f \ e \ l)}_{\substack{\underbrace{\underbrace{: : a} \quad \underbrace{: : b} \quad \underbrace{: : [b]}}_{: : a}}}) \ ls \end{aligned}$$

Es bedeuten:

- ▶  $f$ : Faltungsvorschrift.
- ▶  $e$ : Auffangwert, Vorgabewert für leere Argumentliste.
- ▶  $[], (l:ls)$ : Liste zu aggregierender Werte.

# Aggregieren: Das Funktional foldl (2)

## Anwendungsbeispiele:

```
foldl (+) 0 [] ->> 0
```

```
foldl (+) 0 [2,4..10]
```

```
->> ((+) ((+) ((+) ((+) ((+) 0 2) 4) 6) 8) 10)
```

```
->> (((((0 + 2) + 4) + 6) + 8) + 10) ->> 30
```

```
foldl (*) 1 [] ->> 1
```

```
foldl (*) 1 [2,4..10]
```

```
->> ((*) ((*) ((*) ((*) ((*) 1 2) 4) 6) 8) 10)
```

```
->> (((((1 * 2) * 4) * 6) * 8) * 10) ->> 3.840
```

```
foldl (||) False [] ->> False
```

```
foldl (||) False [True,False,False]
```

```
->> ((||) ((||) ((||) False True) False) False)
```

```
->> (((False || True) || False) || False) ->> True
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

827/170

# Aggregieren: Das Funktional foldl (3)

Anwendungsbeispiele (fgs.): Alternative Definitionen einiger Standardfunktionen in Haskell mittels `foldl`:

```
sum :: Num a => [a] -> a
```

```
sum ns = foldl (+) 0 ns
```

```
prod :: Num a => [a] -> a
```

```
prod ns = foldl (*) 1 ns
```

```
and :: [Bool] -> Bool
```

```
and bs = foldl (&&) True bs
```

```
or :: [Bool] -> Bool
```

```
or bs = foldl (||) False bs
```

```
concat :: [[a]] -> [a]
```

```
concat xss = foldl (++) [] xss
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

828/1700

# foldr, foldl im Vergleich

**foldr**: Falten, zusammenfassen von rechts:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []      = e
foldr f e (l:ls) = f l (foldr f e ls)

foldr f e [a1,a2,...,an]
->> a1 'f' (a2 'f' ... 'f' (an-1 'f' (an 'f' e))...)
```

**foldl**: Falten, zusammenfassen von links:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f e []      = e
foldl f e (l:ls) = foldl f (f e l) ls

foldl f e [b1,b2,...,bn]
->> (...((e 'f' b1) 'f' b2) 'f' ... 'f' bn-1) 'f' bn
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

# Warum zwei Faltungsfunktionale?

...die Funktionele `foldr` und `foldl` unterscheiden sich in

- ▶ Anwendbarkeit
- ▶ Effizienz

abhängig vom [Anwendungskontext](#).

Zur Illustration betrachten wir die Implementierungen der Funktionen `reverse` und `concat` aus dem [Präludium](#):

```
reverse :: [a] -> [a]
reverse = foldl (flip (::)) []
  where flip :: (a -> b -> c) -> (b -> a -> c)
        flip f x y = f y x
```

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

# Zur Anwendbarkeit von `foldl`, `foldr`

Die Implementierung von `reverse` aus dem `Präludium`:

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
  where flip :: (a -> b -> c) -> (b -> a -> c)
        flip f x y = f y x
```

...leistet die gewünschte `Listenumkehrung`; sie hat dieselbe `Bedeutung` wie folgende rekursive Implementierung:

```
reverse :: [a] -> [a]
reverse []      = []
reverse (l:ls) = (reverse ls) ++ [l]

reverse []      ->> []
reverse [1,2,3] ->> [3,2,1]
```

# Zur Wirkung von foldl

...im Zusammenspiel mit der Funktion `reverse`:

```
reverse :: [a] -> [a]
reverse = foldl (flip (::)) []
  where flip :: (a -> b -> c) -> (b -> a -> c)
        flip f x y = f y x
```

...für die Argumentlisten `[]` und `[1,2,3]`:

```
reverse [] ->> foldl (flip (::)) [] [] ->> []
reverse [1,2,3]
->> foldl (flip (::)) [] [1,2,3]
->> ((flip (::)) ((flip (::)) ((flip (::)) [] 1) 2) 3)
->> ((([] 'flip (::)' 1) 'flip (::)' 2) 'flip (::)' 3)
->> (((1 : []) 'flip (::)' 2) 'flip (::)' 3)
->> ((2 : (1 : [])) 'flip (::)' 3)
->> (3 : (2 : (1 : [])))
->> [3,2,1]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

832/170



# Zur Wirkung von foldr

..im Zusammenspiel mit der Funktion `reverse`:

```
rev_untauglich :: [a] -> [a]
rev_untauglich = foldr (flip (:)) []
  where flip :: (a -> b -> c) -> (b -> a -> c)
        flip f x y = f y x
```

...für die Argumentlisten `[]` und `[1,2,3]`:

```
rev_untauglich [] ->> foldr (flip (:)) [] [] ->> []
rev_untauglich [1,2,3]
->> foldr (flip (:)) [] [1,2,3]
->> ((flip (:)) 1 ((flip (:)) 2 ((flip (:)) 3 [])))
->> (1 'flip (:)' (2 'flip (:)' (3 'flip (:)' [])))
->> (1 'flip (:)' (2 'flip (:)' (3 'flip (:)' [])))
->> (1 'flip (:)' (2 'flip (:)' ([ : 3]))
->> Typunverträglichkeit d. Operanden im Term ([ : 3).
Auswertungsversuch von rev_untauglich scheitert!
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

833/170

## Übungsaufgabe 10.5.3.1: $(:)$ statt $(\text{flip } (:))$

Vollziehe ebenfalls mit Papier und Bleistift nach (z.B. für die Argumentliste  $[1,2,3]$ ), dass sich die Faltungsfunktionale  $\text{foldl}$  und  $\text{foldr}$  auch bezüglich  $(:)$  als Faltungsoperation unterschiedlich verhalten. Zeige dazu, dass folgender Versuch

- ▶  $\text{untauglich}$  ist: Auswertungsversuche für nichtleere Listen scheitern aufgrund von Operandenunverträglichkeiten:

```
rev_untauglich' :: [a] -> [a]
rev_untauglich' = foldl (:) []
```

- ▶ die Identität auf Listen liefert:

```
rev_id :: [a] -> [a]
rev_id = foldr (:) []
```

d.h. für alle Listen  $xs$  gilt:  $\text{rev\_id } xs \rightarrow xs$

# Zur Effizienz von `concat`, `slow_concat` (1)

Vergleiche die **Effizienz**, **Performanz** von:

- `concat` wie im **Präludium** mittels `foldr` definiert:

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

...mit **derjenigen** von:

- `slow_concat` mittels `foldl` definiert:

```
slow_concat :: [[a]] -> [a]
slow_concat = foldl (++) []
```

## Zur Effizienz von concat, slow\_concat (2)

...seien (vereinfachend) alle Listen  $xs_i$  v. gleicher Länge  $l$ .

Dann hängen die (Kopier-) Kosten der Berechnung von

```
► concat [xs1,xs2,...,xsn]
  ->> foldr (++) [] [xs1,xs2,...,xsn]
  ->> xs1 ++ (xs2 ++ (... (xsn ++ [])) ...)
```

linear von der Anzahl  $n$  der Listen  $xs_i$  ab:  $n * l$  (jedes Konkatenieren erfolgt an eine Präfixliste der Länge  $l$ ); die von

```
► slow_concat [xs1,xs2,...,xsn]
  ->> foldl1 (++) [] [xs1,xs2,...,xsn]
  ->> (... (([] ++ xs1) ++ xs2) ...) ++ xsn
```

hingegen quadratisch:  $n * (n - 1) * l$  (das Konkatenieren erfolgt an sukzessive länger werdende Präfixlisten:  $0, l, (l + l), (l + l + l), \dots, (n - 1) * l$ ).

# Zur Effizienz von concat, slow\_concat (3)

...wobei  $n * (n - 1) * l$  Abschätzung ist der Summe:

$$\begin{aligned} & 0 \\ & + l \\ & + (l + l) \\ & + (l + l + l) \\ & \dots \\ & + \underbrace{(l + l + \dots + l)}_{(n-1)\text{-mal}} \\ & = \sum_{i=1}^{n-1} i * l \\ & = \left( \sum_{i=1}^{n-1} i \right) * l \\ & = \frac{n * (n - 1)}{2} * l \end{aligned}$$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

837/170

# Übungsaufgabe 10.5.3.2

Untersuche und vergleiche auch die **Effizienz** und **Performanz** der rekursiven Implementierung von **reverse**:

```
reverse :: [a] -> [a]
reverse []      = []
reverse (l:ls) = (reverse ls) ++ [l]
```

...mit der **Effizienz** und **Performanz** der Implementierung aus dem **Präludium**:

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
  where flip :: (a -> b -> c) -> (b -> a -> c)
        flip f x y = f y x
```

# Zusammenfassung

...die Beispiele zeigen, dass sich die Faltungsfunktionale `foldr` und `foldl` unterscheiden können hinsichtlich

- ▶ Anwendbarkeit (`foldr`, `foldl` mit Faltungsfunktionen `(flip (:))`, `(:)`)
- ▶ Effizienz (`foldr`, `foldl` mit Faltungsfunktion `(++)`)

...Eignung und Wahl von `foldr` und `foldl` sind deshalb **problem-** und **kontextabhängig** festzustellen und zu treffen!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.5.1

839/170

# Kapitel 10.6

## Beispiel: Rechnen mit Funktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

**10.6**



# Rechnen mit Funktionen

...am Beispiel des Algorithmus' von Euklid zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen.

Der Algorithmus von Euklid zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen  $m, n \in \mathbb{N}_1$  ist wie folgt:

- ▶ Wähle  $x$  gleich  $m$  und  $y$  gleich  $n$ .
- ▶ Ziehe wiederholt den kleineren der Werte von  $x$  und  $y$  vom größeren ab.
- ▶ Höre auf, wenn  $x$  und  $y$  denselben Wert haben. Dieser Wert ist der größte gemeinsame Teiler von  $m$  und  $n$ .

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

841/170

# Applikativ: Rechnen mit elementaren Werten

```
type Nat1 = Integer
```

```
ggt_euklid_app :: Nat1 -> Nat1 -> Nat1
```

```
ggt_euklid_app x y
```

```
  | x > y  = ggt_euklid_app (x-y) y
```

```
  | x < y  = ggt_euklid_app x (y-x)
```

```
  | x == y = x
```

Zwei **Aufrufbeispiele** zur Illustration der Arbeitsweise von **ggt\_euklid\_app**:

```
m = 18; n = 12
```

```
ggt_euklid_app m n ->> ggt_euklid_app 18 12
```

```
->> ggt_euklid_app 6 12 ( $\hat{=}$  18-12 12)
```

```
->> ggt_euklid_app 6 6 ( $\hat{=}$  6 12-6) ->> 6
```

```
m' = 20; n' = 35
```

```
ggt_euklid_app m' n' ->> ggt_euklid_app 20 35
```

```
->> ggt_euklid_app 20 15 ( $\hat{=}$  20 35-20)
```

```
->> ggt_euklid_app 5 15 ( $\hat{=}$  20-15 15)
```

```
->> ggt_euklid_app 5 10 ( $\hat{=}$  5 15-5)
```

```
->> ggt_euklid_app 5 5 ( $\hat{=}$  5 10-5) ->> 5
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

842/170

# Funktional: Rechnen mit Funktionen (1)

```
type Nat1      = Integer
data Variable  = X | Y deriving (Eq,Show)
type Variablen = Variable
type Zustand   = (Variablen -> Nat1)
type Sigma     = Zustand

ggt_euklid_fkt :: Sigma -> Sigma
-- ggt_euklid_fkt :: (Variablen -> Nat1) -> (Variablen -> Nat1)
ggt_euklid_fkt sigma
  | sigma X > sigma Y
    = ggt_euklid_fkt (\z -> if z==X then sigma X - sigma Y
                          else sigma Y)
  | sigma X < sigma Y
    = ggt_euklid_fkt (\z -> if z==X then sigma X
                          else sigma Y - sigma X)
  | sigma X == sigma Y = sigma

ggt :: Nat1 -> Nat1 -> Nat1
ggt m n = (ggt_euklid_fkt (\z -> if z==X then m else n)) X
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

843/170

# Funktional: Rechnen mit Funktionen (2)

Zwei [Aufrufbeispiele](#) zur Illustration der Arbeitsweise von `ggt` und `ggt_euklid_fkt`:

```
m = 18
n = 12
ggt m n
->> (ggt_euklid_fkt sigma1) X
      where sigma1 X = 18
          sigma1 Y = 12
->> (ggt_euklid_fkt sigma2) X
      where sigma2 X = 6 ( $\hat{=}$  sigma1 X - sigma1 Y ->> 18 - 12 ->> 6)
          sigma2 Y = 12 ( $\hat{=}$  sigma1 Y ->> 12)
->> (ggt_euklid_fkt sigma3) X
      where sigma3 X = 6 ( $\hat{=}$  sigma2 X ->> 6)
          sigma3 Y = 6 ( $\hat{=}$  sigma2 Y - sigma2 X ->> 12 - 6 ->> 6)
->> sigma3 X
->> 6
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

844/170

# Funktional: Rechnen mit Funktionen (3)

$m' = 20$

$n' = 35$

$\text{ggT } m' \ n'$

```
->> (ggT_euklid_fkt sigma1) X
      where sigma1 X = 20
            sigma1 Y = 35
->> (ggT_euklid_fkt sigma2) X
      where sigma2 X = 20 ( $\hat{=}$  sigma1 X ->> 20)
            sigma2 Y = 15 ( $\hat{=}$  sigma1 Y - sigma1 X ->> 35-20 ->> 15)
->> (ggT_euklid_fkt sigma3) X
      where sigma3 X = 5  ( $\hat{=}$  sigma2 X - sigma2 Y ->> 20-15 ->> 5)
            sigma3 Y = 15 ( $\hat{=}$  sigma2 Y ->> 15)
->> (ggT_euklid_fkt sigma4) X
      where sigma4 X = 5  ( $\hat{=}$  sigma3 X ->> 5)
            sigma4 Y = 10 ( $\hat{=}$  sigma3 Y - sigma3 X ->> 15-5 ->> 10)
->> (ggT_euklid_fkt sigma5) X
      where sigma5 X = 5  ( $\hat{=}$  sigma4 X ->> 5)
            sigma5 Y = 5  ( $\hat{=}$  sigma4 Y - sigma4 X ->> 10-5 ->> 5)
->> sigma5 X
->> 5
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

# Beachte: Applikativität von `ggt_euklid_app`

...formal argumentiert ist auch die Funktion `ggt_euklid_app` eine Funktion, die eine **Funktion als Ergebnis** hat und insofern mit '**Funktionen rechnet**' wie die explizite Klammerung der Typsignatur:

```
ggt_euklid_app :: Nat1 -> (Nat1 -> Nat1)
```

und ein Aufruf wie:

```
ggt_euklid_app 42 :: (Nat1 -> Nat1)
```

zeigen.

Die Einführung des uncurryfizierten applikativen Gegenstücks `ggt_euklid_app'` zu `ggt_euklid_app` zeigt, dass das Argument auf der formalen Ebene bleibt und `ggt_euklid_app` im Kern 'applikativ', nicht 'echt funktional' ist:

```
ggt_euklid_app' :: (Nat1,Nat1) -> Nat1  
ggt_euklid_app' (m,n) = ggt_euklid_app m n
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

# Beachte: Funktionalität von `ggt_euklid_fkt`

Im Unterschied dazu ist `ggt_euklid_fkt` eine Funktion, die im wahren Sinn mit 'Funktionen rechnet', nämlich Funktionen auf Funktionen abbildet:

```
ggt_euklid_fkt :: (Variablen -> Nat1) -> (Variablen -> Nat1)
```

wie die Expansion der Typsignatur verdeutlicht.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

847/170

# Übungsaufgabe 10.6.1

Am Anfang von [Teil III](#), unmittelbar vor [Kapitel 7](#), ist eine Charakterisierung und Abgrenzung [applikativer](#) und [funktionaler Programmierung](#) gegeben.

Vollziehen Sie diese Charakterisierung und Abgrenzung am Beispiel der Funktionen [ggt\\_euklid\\_app](#) und [ggt\\_euklid\\_fkt](#) (so weit als durch die Einfachheit dieses Beispiels möglich) nach.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

848/170



# Kapitel 10.7

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

# Zusammenfassung

...Programmierung mit **Funktionalen** macht

- ▶ das **Wesen funktionaler Programmierung** aus.

...unterstützt insbesondere

- ▶ **Wiederverwendung** von Programmcode.
- ▶ **Kürzere** und meist **einfacher zu verstehende** Programme.
- ▶ **Einfachere Herleitung**, **einfacherer Beweis** von Programmeigenschaften (Stichwort: **Programmverifikation**).
- ▶ ...

...vordefinierte **Funktionale auf Listen** leisten einen wesentlichen Beitrag hierzu.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

# Kapitel 10.8

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 10 (1)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 6, Funktionen höherer Ordnung)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 5, Listen und Funktionen höherer Ordnung)
-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 5, Polymorphic and Higher-Order Functions; Kapitel 9, More about Higher-Order Functions)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2





10.3

10.4

10.5

10.6

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 10 (2)

-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 7, Higher-order functions)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 5, Higher-order Functions)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 4, Functional Programming)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 8, Funktionen höherer Ordnung)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2


10.3


10.4


10.5

10.6

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 10 (3)

 Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 2.5, Higher-order functional programming techniques)

 Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 9.2, Higher-order functions: functions as arguments; Kapitel 10, Functions as values; Kapitel 19.5, Folding revisited)

 Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 11, Higher-order functions; Kapitel 12, Developing higher-order programs; Kapitel 20.5, Folding revisited)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

10.1

10.2

10.3

10.4

10.5

10.6

# Kapitel 11

## Polymorphie

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

**Kap. 11**

11.1

11.2

11.3

11.4

855/170

# Polymorphie

Grundbedeutung lt. Duden:

- ▶ Vielgestaltigkeit, Verschiedengestaltigkeit

...mit verschiedenen fachspezifischen **Bedeutungsausprägungen**:

- ▶ **Chemie**: Vorkommen mancher Mineralien in unterschiedlicher Form, mit unterschiedlichen Eigenschaften, aber gleicher chemischer Zusammensetzung.
- ▶ **Biologie**: Vielgestaltigkeit der Blätter oder der Blüte einer Pflanze.
- ▶ **Sprachwissenschaft**: Vorhandensein mehrerer sprachlicher Formen für den gleichen Inhalt, die gleiche Funktion (z.B. verschiedenartige Pluralbildungen in: die Tiere, die Felder, die Wiesen, die Pontons).
- ▶ **Informatik**, speziell **Theorie der Programmiersprachen**: Vieltypigkeit.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4



# Kapitel 11.1

## Motivation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Im programmiersprachlichen Kontext

...unterscheiden wir **Polymorphie** auf:

- ▶ **Datentypen** (Kap. 11.2)

- ▶ Algebraische Datentypen, `data`
- ▶ Neue Typen, `newtype`
- ▶ Typsynonyme, `type`

⇒ Sprachmittel: **Typvariablen**, **Typklassen**.

- ▶ **Funktionen** (Kap. 11.3, 11.4)

- ▶ Parametrische Polymorphie (oder **echte Polymorphie**)

⇒ Sprachmittel: **Typvariablen**.

- ▶ *Ad hoc* Polymorphie (oder **unechte Polymorphie**, Überladung)

⇒ Haskell-spezifisches Sprachmittel: **Typklassen**.

# Typvariablen, Typklassen in Haskell

Typvariablen: Freiwählbare Identifikatoren als Bezeichnungen

- ▶ beginnend mit einem Kleinbuchstaben, üblicherweise vom Anfang des Alphabets gewählt (z.B.: `a`, `b`, `fp185A03`,...)
- ▶ mit Typen als Wert.

Typklassen: Freiwählbare Identifikatoren als Bezeichnungen

- ▶ beginnend mit einem Großbuchstaben (z.B.: `Eq`, `Ord`, `Analysierbar`, `Warnung`,...)
- ▶ mit Typen als Instanzen.

Bem: Bezeichnungen für Typnamen, Typ- und Datenwertkonstruktoren sind in Haskell ebenfalls freiwählbare Identifikatoren, die wie die von Typklassen mit einem

- ▶ Großbuchstaben beginnen müssen (z.B.: `A`, `B`, `True`, `False`, `String`, `Blatt`, `Wurzel`, `FP185A03`,...).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

859/1700

# Kapitel 11.2

## Polymorphie auf Datentypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

**11.2**

11.2.1

11.2.2

# Polymorphe Datentypen

## Definition 11.2.1 (Polymorpher Datentyp)

Ein algebraischer (Daten-) Typ, neuer Typ oder Typsynonym  $T$  heißt **polymorph**, wenn einer oder mehrere Grundtypen der Werte von  $T$  in Form einer oder mehrerer **Typvariablen** als Typparameter angegeben werden.

Beispiele **polymorpher Datentyp(declaration)**en:

```
data Baum a b c = Blatt a b
                  | Wurzel (Baum a b c) c (Baum a b c)

newtype Paartripel a b c d = Pt ((a,b),(b,c),(c,d))

type Assoziationsliste a b = [(a,b)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

861/170

# Kapitel 11.2.1

## Polymorphe algebraische Datentypen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

**11.2.1**

11.2.2

# Bsp. polymorpher algebraischer Datentypen

...Listen, Bäume, Graphen, gewichtete Graphen:

```
type Gewicht = Int
```

```
-- Ohne Kontexteinschränkung:
```

```
data Liste a = Leer
```

```
          | Kopf a (Liste a)
```

```
data Baum a b c = Blatt a b
```

```
          | Wurzel (Baum a b c) c (Baum a b c)
```

```
data Graph a = Gph (a -> [a])
```

```
data GewichteterGraph a = GGph (a -> [(a,Gewicht)])
```

```
-- Mit Kontexteinschränkung:
```

```
data Eq a => Liste' a = Leer'
```

```
          | Kopf' a (Liste' a)
```

```
data (Eq a, Ord b, Ord c, Num c) => Baum' a b c
```

```
  = Blatt' a b
```

```
    | Wurzel' (Baum' a b c) c (Baum' a b c)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

863/170

# Beispiele gültiger Listen- und Baumwerte

-- Listenwerte:

Leer :: Liste a

Kopf 17 (Kopf 4 (Kopf (17+4) Leer)) :: Liste Int

Kopf 'a' (Kopf 'e' (Kopf 'i' (Kopf 'o' (Kopf 'u' Leer))))  
:: Liste Char

Kopf True (Kopf (True&&False) (Kopf ((odd.fib) 42) Leer))  
:: Liste Bool

-- Baumwerte:

Blatt "Fun Prog" 8 :: Baum [Char] Int c

Blatt True 3.14 :: Baum Bool Float c

Blatt 'a' 'z' :: Baum Char Char c

Wurzel (Blatt "Fun" 3) True (Blatt "Prog" 4)  
:: Baum [Char] Int Bool

Wurzel (Blatt "Fun" 3) (Kopf 42 Leer) (Blatt "Prog" 4)  
:: Baum [Char] Int (Liste Int)

Wurzel (Blatt "Fun" 3) Leer (Blatt "Prog" 4)  
:: Baum [Char] Int (Liste a)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2



# Beispiele gültiger Graph- und gew. Graphwerte

-- Graphwerte:

```
data Knoten = K1 | K2 | K3 deriving (Eq,Show)
g :: Knoten -> [Knoten]
g K1 = [K1,K2,K3]
g K2 = [K2,K3]
g K3 = []
g' :: Int -> [Int]
g' = \n -> [n..2*n]           -- gleichbed.: g' n = [n..2*n]
Gph g :: Graph Knoten
Gph g' :: Graph Int
```

-- Gewichtete Graphwerte:

```
gg :: Knoten -> [(Knoten,Gewicht)]
gg K1 = [(K1,0),(K2,17),(K3,4)]
gg K2 = [(K2,0),(K3,42)]
gg K3 = []
gg' :: Int -> [(Int,Gewicht)]
gg' = \n -> [(m,m+1) | m <- [n..2*n]]
GGph gg :: GewichteterGraph Knoten
GGph gg' :: GewichteterGraph Int
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

865/170

# Kapitel 11.2.2

## Polymorphe neue Typen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

# Beispiele polymorpher neuer Typen

...Paare, Tripel, Graphen, Relationen, Funktionen u.a.:

-- Ohne Kontexteinschränkung:

```
newtype Unverwechselbar_mit_Typ_a a = Uvb a
```

```
newtype Paar a = P (a,a)
```

```
newtype Tripelpaar a b c d = TP ((a,b),(b,c),(c,d))
```

```
newtype Graph' a = Gph' (a -> [a])
```

```
newtype Relation a b = R [(a,b)]
```

```
newtype Funktion a b = F (a->b)
```

-- Mit Kontexteinschränkung:

```
newtype Ord a => Paar' a = P' (a,a)
```

```
newtype (Ord a, Ord b) => Relation' a b = R' [(a,b)]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2.1

11.2.2

867/170

# Beispiele gültiger Paar- und Tripelpaarwerte

-- Unverwechselbare Werte:

Uvb 4711 :: Unverwechselbar\_mit\_Typ\_a Int

Uvb Leer :: Unverwechselbar\_mit\_Typ\_a (Liste a)

Uvb sqrt :: Unverwechselbar\_mit\_Typ\_a (Float -> Float)

-- Paarwerte:

P (17,4) :: Paar Int

P ([],[42]) :: Paar [Int]

P ([],[a]) :: Paar [a]

-- Tripelpaarwerte:

TP (("Fun",3),(4,odd(length "Prog")), (True, 'T'))  
:: Unverwechselbar\_mit\_Typ\_a [Char] Int Bool Char

TP ((fac, 'a'), ('z', ""), ("Hallo, Welt!", id) )  
:: Unverwechselbar\_mit\_Typ\_a (Integer -> Integer)  
Char [Char] (a -> a)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

868/170

# Beispiele gültiger Relations-, Funktionswerte

-- Graphwerte:

Gph' (\c -> ['a'..c]) :: Graph' Char

Gph' (\n -> [0..n]) :: Graph' Int

Gph' (\b -> [not b]) :: Graph' Bool

-- Relationswerte:

R [(1,1),(1,3),(2,2),(2,3),(3,3)] :: Relation Int Int

R [(Leer,42)] :: Relation (Liste a) Int

R [] :: Relation a b

-- Funktionswerte:

F fac :: Funktion Integer Integer

F id :: Funktion a a

F (\x->(\y->(x > length y))) :: Funktion Int ([a] -> Bool)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

869/170

# Kapitel 11.2.3

## Polymorphe Typsynonyme

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

# Beispiele polymorpher Typsynonyme

...Sequenzen, Assoziationslisten u.a.:

-- Ohne Kontexteinschränkung:

```
type Sequenz a = [a]
```

```
type Ass_Liste a b = [(a,b)]
```

```
type MeinTyp a = Unverwechselbar_mit_Typ_a a
```

```
type MeinBaum a b c = Baum a b c
```

```
type MeinPaar a b = (Funktion a b, Relation a b)
```

...Einschränkungen durch Kontexte für Typalias nicht erlaubt.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

# Beispiele gültiger Sequenz-, Ass.-Listenwerte

## -- Sequenzwerte

```
leereSequenz = [] :: Sequenz a
intSequenz = [1,2,3,4,6,12] :: Sequenz Int
fktSequenz = [fac,fib,(+1)] :: Sequenz (Integer -> Integer)

Main>:t leereSequenz ->> [a]
Main>:t intSequenz ->> [Int]
Main>:t fktSequenz ->> [Integer -> Integer]
```

## -- Assoziationslistenwerte

```
leereAssListe = [] :: Ass_Liste a b
al = [("Hallo,",6),(" ",1),("Welt!",5)]
                                     :: Ass_Liste String Int
al' = [(fac,"fac"),(fib,"fib"),((+1),"inc")]
                                     :: Ass_Liste (Integer -> Integer) String

Main>:t leereAssListe ->> [(a,b)]
Main>:t al ->> [( [Char],Int)]
Main>:t al' ->> [(Integer -> Integer, [Char])]
```

**Beachte:** Die Typen werden zum Grund-, nicht zum Aliastyp aufgelöst (z.B. für intSequenz Typ `[Int]` statt Typ `Sequenz Int`).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

872/1700



# Beispiele gültiger Mein-'xy'-Werte

-- MeinTyp-Werte

zahlwert = Uvbmta 4711 :: MeinTyp Int

lst\_wert = Uvbmta Leer :: MeinTyp (Liste a)

fkt\_wert = Uvbmta sqrt :: MeinTyp (Float -> Float)

Main>:t zahlwert ->> Unverwechselbar\_mit\_Typ\_a Int

Main>:t lst\_wert ->> Unverwechselbar\_mit\_Typ\_a (Liste a)

Main>:t fkt\_wert

->> Unverwechselbar\_mit\_Typ\_a (Float -> Float)

-- MeinBaum-Wert

baumwert = Blatt "Fun Prog" 8 :: MeinBaum [Char] Int Bool

Main>:t baumwert ->> Baum [Char] Int c

-- MeinPaar-Wert

paarwert = (F (+1),R []) :: MeinPaar Int Int

Main>:t paarwert ->> (Funktion Int Int,Relation Int Int)

Beachte wieder die Auflösung zum Grund-, nicht zum Aliastyp.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

873/170

# Kapitel 11.2.4

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

# Zusammenfassung

Polymorphie auf Datentypen erlaubt Wiederverwendung von

- Datenstrukturnamen (Typ- und Konstruktornamen)  
(Gute Namen sind knapp!)

deren Werte strukturell gleich aufgebaut sind und sich nur die Typen ihrer Wertbenennungen unterscheiden.

So sind die drei Baumtypen:

```
Baum Int Int Int  
Baum Int Char String  
Baum (a->c) (b->c) (a->b)
```

deren Werte sich nur durch die Wertbenennung unterscheiden, alle Instanzen desselben polymorphen Typs:

```
data Baum a b c = Blatt a b  
                  | Wurzel (Baum a b c) c (Baum a b c)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.2.1

11.2.2

# Übungsaufgabe 11.2.4.1

Ergänze Beispiele gültiger Werte für folgende Typen:

► Kapitel 11.2.1:

Liste' a

Baum' a b c

► Kapitel 11.2.2:

Paar' a

Relation' a b

► Kapitel 11.2.1, 11.2.2 und 11.2.3:

MeinBaum' a b c

MeinPaar' a b

mit

type MeinBaum' a b c = Baum' a b c

type MeinPaar' a b = Paar' a b

# Kapitel 11.3

## Parametrische Polymorphie auf Funktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

**11.3**

11.3.1

# Parametrisch polymorphe Funktionen

## Definition 11.3.1 (Parametrisch polymorphe Fkt.)

Eine Funktion heißt **parametrisch polymorph** (oder **echt polymorph**), wenn die Typen eines oder mehrerer ihrer Parameter angegeben durch **Typvariablen** Werte beliebiger Typen als Argument zulassen.

Beispiele **parametrisch polymorpher** Funktionen:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
```

```
curry f x y = f (x,y)
```

```
fst :: (a,b) -> a
```

```
fst (x,_) = x
```

```
length :: [a] -> Int
```

```
length [] = 0
```

```
length (_:xs) = 1 + length xs
```

```
splitAt :: Int -> [a] -> ([a],[a])
```

```
splitAt n xs = (take n xs, drop n xs)
```

# Parametrische Polymorphie auf Funktionen

...tritt in funktionalen Sprachen **beiläufig** und **ubiquitär** auf wie das Beispiel vieler vordefinierter Funktionen zeigt:

- ▶ Die Funktionale **curry**, **uncurry**, **flip** und **id**.
- ▶ Die Funktionale **map**, **filter**, **foldl** und **foldr**.
- ▶ Die Funktionen **fst** und **snd**.
- ▶ Die Funktionen **length**, **head** und **tail**.
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

# Kapitel 11.3.1

## Vordefinierte parametrisch polymorphe Funktionen



# Die parametrisch polymorphen Funktione

...curry, uncurry, flip und id auf beliebigen Typen:

$\text{curry} :: ((a, b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$

$\text{curry } f \ x \ y = f \ (x, y)$

$\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow ((a, b) \rightarrow c)$

$\text{uncurry } g \ (x, y) = g \ x \ y$

$\text{flip} :: (a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c)$

$\text{flip } f \ x \ y = f \ y \ x$

$\text{id} :: a \rightarrow a$

$\text{id } x = x$

Anwendungsbeispiele:

$\text{id } 3 \rightarrow 3$

$\text{id } ["abc", "def"] \rightarrow ["abc", "def"]$

$\text{id } \text{fac } 5 \rightarrow (\text{id } \text{fac}) \ 5 \rightarrow \text{fac } 5 \rightarrow 120$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

881/170

# Die parametrisch polymorphen Funktionale

...map, filter, foldl und foldr auf beliebigen Listentypen:

```
map :: (a -> b) -> [a] -> [b]
```

```
map f xs = [f x | x <- xs]
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter p xs = [x | x <- xs, p x]
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr f e [] = e
```

```
foldr f e (x:xs) = f x (foldr f e xs)
```

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

```
foldl f e [] = e
```

```
foldl f e (x:xs) = foldl f (f e x) xs
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

882/170

# Die parametrisch polymorphen Funktionen

...`fst` und `snd` auf beliebigen Paartypen:

```
fst :: (a,b) -> a
```

```
fst (x,_) = x
```

```
snd :: (a,b) -> b
```

```
snd (_,y) = y
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

# Die parametrisch polymorphen Funktionen

...length, head und tail auf beliebigen Listentypen:

```
length :: [a] -> Int
length []      = 0
length (_:xs) = 1 + length xs

head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

# Die parametrisch polymorphen Funktione

...zip und unzip ('Reißverschlussfunktionen') auf beliebigen Listentypen:

```
zip :: [a] -> [b] -> [(a,b)]
```

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

```
zip _ _          = []
```

```
zip [3,4,5] ['a','b','c','d'] ->> [(3,'a'),(4,'b'),(5,'c')]
```

```
zip ["abc","def","geh"] [(3,4),(5,4)]
```

```
->> [("abc",(3,4)),("def",(5,4))]
```

```
unzip :: [(a,b)] -> ([a],[b])
```

```
unzip []          = ([],[])
```

```
unzip ((x,y):ps) = (x:xs,y:ys)
```

```
    where
```

```
        (xs,ys) = unzip ps
```

```
unzip [(3,'a'),(4,'b'),(5,'c')] ->> ([3,4,5],['a','b','c'])
```

```
unzip [("abc",(3,4)),("def",(5,4))]
```

```
->> (["abc","def"],[(3,4),(5,4)])
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

885/170

# Weitere vordefinierte parametrisch polymorphe

...Funktionale und Funktionen auf beliebigen Listentypen:

<code>(:)</code>	<code>::</code>	<code>a -&gt; [a] -&gt; [a]</code>	Listenkonstruktor (rechtsassoziativ)
<code>(!!)</code>	<code>::</code>	<code>[a] -&gt; Int -&gt; a</code>	Projektion auf i-te Komp., Infixop.
<code>length</code>	<code>::</code>	<code>[a] -&gt; Int</code>	Länge der Liste
<code>(++)</code>	<code>::</code>	<code>[a] -&gt; [a] -&gt; [a]</code>	Konkat. zweier Listen
<code>concat</code>	<code>::</code>	<code>[[a]] -&gt; [a]</code>	Konkat. mehrerer Listen
<code>head</code>	<code>::</code>	<code>[a] -&gt; a</code>	Listenkopf
<code>last</code>	<code>::</code>	<code>[a] -&gt; a</code>	Listenendelement
<code>tail</code>	<code>::</code>	<code>[a] -&gt; [a]</code>	Liste ohne Listenkopf
<code>init</code>	<code>::</code>	<code>[a] -&gt; [a]</code>	Liste ohne Endelement
<code>splitAt</code>	<code>::</code>	<code>Int -&gt; [a] -&gt; ([a], [a])</code>	Aufspalten einer Liste an Position i
<code>reverse</code>	<code>::</code>	<code>[a] -&gt; [a]</code>	Umkehren einer Liste
...			

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

886/170

# Kapitel 11.3.2

## Selbstdefinierte parametrisch polymorphe Funktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

887/170

# Parametrisch polymorphe Funktionen

...auf (selbstdefinierten) algebraischen Datentypen:

```
data Baum a b c = Blatt a b
                  | Wurzel (Baum a b c) c (Baum a b c)

tiefe :: (Baum a b c) -> Int
tiefe (Blatt _ _) = 0
tiefe (Wurzel ltb _ rtb) = 1 + max (tiefe ltb) (tiefe rtb)

gen_assliste :: (Baum a b c) -> Ass_Liste a b
gen_assliste (Blatt x y)      = [(x,y)]
gen_assliste (Wurzel ltb _ rtb) = (gen_assliste ltb)
                                   ++ (gen_assliste rtb)

plaetten :: (a -> b -> c) -> (Baum a b c) -> (Sequenz c)
plaetten f (Blatt x y)      = [f x y]
plaetten f (Wurzel ltb z rtb) = (plaetten f ltb)
                                   ++ [z]
                                   ++ (plaetten f rtb)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

888/170



# Einschränkung parametrischer Polymorphie

...durch **Typkontexte** bei der **Funktionsdefinition**:

```
data Baum a b c = Blatt a b
                  | Wurzel (Baum a b c) c (Baum a b c)

wurzelsumme :: Num c => (Baum a b c) -> c
wurzelsumme (Blatt _ _) = 0
wurzelsumme (Wurzel ltb z rtb)
  = z + wurzelsumme ltb + wurzelsumme rtb
```

...bereits bei der **Datentypdeklaration**:

```
data Num c => Baum' a b c = Blatt' a b
              | Wurzel' (Baum' a b c) c (Baum' a b c)

wurzelsumme' :: Num c => (Baum' a b c) -> c
wurzelsumme' (Blatt' _ _) = 0
wurzelsumme' (Wurzel' ltb z rtb)
  = z + wurzelsumme' ltb + wurzelsumme' rtb
```

wobei der **Kontext** (**Num c**) auch bei **wurzelsumme'** nötig ist.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

# Bemerkungen zur Typkontexteinschränkung

Die **Einschränkung parametrischer Polymorphie** auf numerische Typen als zulässige Instanzen der Typvariable **c** in der Signatur der Funktionen

► `wurzelsumme`, `wurzelsumme'`

ist nötig, weil sich

► beide Funktionen auf die (überladene) Funktion **(+)** abstützen, die ausschließlich für Werte numerischer Typen definiert ist, d.h. für Werte von Typen, die Element der Typklasse **Num** sind (siehe auch [Kap. 11.4](#)).

(**Bem.:** Ohne Kontext wird in `wurzelsumme'` der Typ **Integer** als Typ für **c** inferiert, nicht dass **c** ein Typ in **Num** ist.)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

890/170

# Kapitel 11.3.3

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

# Zusammenfassung

Parametrische Polymorphie auf Funktionen ermöglicht **Wiederverwendung** von

- ▶ Funktionsnamen (**Gute Namen sind knapp!**)
- ▶ Funktionsrümpfen (**Implementierungen sind aufwändig!**)

für beliebige Typen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

# Veranschaulichung von Wiederverwendung (1)

...anhand der Funktion `length` auf beliebigen Listentypen:

```
length :: [a] -> Int
length []      = 0
length (_:xs) = 1 + length xs
```

Dank **parametrischer Polymorphie** sind Aufrufe mit Listen über beliebigen Listenelementtypen unmittelbar möglich, z.B.:

```
length [2,4,23,2,53,4] ->> 6
length ["Enjoy","Functional","Programming"] ->> 3
length [(3.14,42.0),(56.1,51.3),(1.12,2.22)] ->> 3
length [[2,4,23,2,5],[3,4],[],[56,7,6,12]] ->> 4
length [(Blatt 17 4), (Blatt 21 21),
        (Wurzel (Blatt 47 11) fac (Blatt 42 0))] ->> 3
length [fac,fib,fun91,(binom 45),(+1),(*2)] ->> 6
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

893/170

# Veranschaulichung von Wiederverwendung (2)

Ohne parametrische Polymorphie wäre für jeden Listentyp eine eigene Implementierung unter eigenem Namen nötig, die sich einzig in **Namen** und **Typsignatur** unterscheiden:

```
length_Ganzzahlliste :: [Int] -> Int
length_Ganzzahlliste [] = 0
length_Ganzzahlliste (_,xs) = 1 + length_Ganzzahlliste t xs

length_Zeichenreihenliste :: [String] -> Int
length_Zeichenreihenliste [] = 0
length_Zeichenreihenliste (_,xs)
  = 1 + length_Zeichenreihenliste xs

...

length_Baumliste :: [Baum] -> Int
length_Baumliste [] = 0
length_Baumliste (_,xs) = 1 + length_Baumliste xs

length_Funktionsliste :: [(Integer -> Integer)] -> Int
length_Funktionsliste [] = 0
length_Funktionsliste (_,xs) = 1 + length_Funktionsliste xs
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

894/170

# Veranschaulichung von Wiederverwendung (3)

...und **argumenttypspezifische** Funktionsaufrufe erforderten:

```
length_Ganzzahlliste [2,4,23,2,53,4] ->> 6
```

```
length_Zeichenreihenliste  
["Enjoy","Functional","Programming"] ->> 3
```

```
length_Gleitkommazahlpaarliste  
[(3.14,42.0),(56.1,51.3),(1.12,2.22)] ->> 3
```

```
length_Ganzzahllistenliste  
[[2,4,23,2,5],[3,4],[],[56,7,6,12]] ->> 4
```

```
length_Baumliste  
[(Blatt 17 4), (Blatt 21 21),  
 (Wurzel (Blatt 47 11) fac (Blatt 42 0))] ->> 3
```

```
length_Funktionsliste  
[fac,fib,fun91,(binom 45),(+1),(*2)] ->> 6
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

# Sprechweisen: Monomorphie vs. Polymorphie

## Rechenvorschriften

### ► der Form:

- `length_Ganzzahlliste :: [Int] -> Int`
- `length_Zeichenreihenliste :: [String] -> Int`
- `length_Gleitkommazahlpaarliste ::  
[(Float,Float)] -> Int`
- `length_Ganzzahllistenliste :: [[Int]] -> Int`
- `length_Baumliste :: [Baum] -> Int`
- `length_Funktionsliste ::  
[(Integer -> Integer)] -> Int`

heißen **monomorph** (definiert für genau einen Typ).

### ► der Form:

- `length :: [a] -> Int`

heißen **parametisch polymorph** (oder **echt polymorph**)  
(definiert für alle Typen).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.3.1

896/170



# Sprechweisen: Typinstanz, allgemeinsten Typ

...illustriert anhand der Typsignatur der Funktion `length`:

```
length :: [a] -> Int
```

Sprechweisen:

► Typen wie

```
[Int] -> Int
```

```
[String] -> Int
```

```
[(Float,Float)] -> Int
```

```
[(Integer -> Integer)] -> Int
```

...

heißen **Instanzen** des Typs `([a] -> Int)`

- Der Typ `([a] -> Int)` heißt **allgemeinster Typ** der Typen `[Int] -> Int`, `[String] -> Int`, `[(Float,Float)] -> Int`, etc. (s. **Kap. 14**).

# Das Hugs-Kommando :t

...liefert stets den (eindeutig bestimmten) **allgemeinsten** Typ eines wohlgeformten Haskell-Ausdrucks, wobei Typsynonyme zum Grundtyp hin aufgelöst werden (z.B. `[(a,b)]` und `[c]` statt `(Ass_Liste a b)` und `(Sequenz c)`):

## Beispiele:

```
Main>:t length
```

```
length :: [a] -> Int
```

```
Main>:t curry
```

```
curry :: ((a,b) -> c) -> (a -> b -> c)
```

```
Main>:t flip
```

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

```
Main>:t gen_assliste
```

```
gen_assliste :: (Baum a b c) -> [(a,b)]
```

```
Main>:t plaetten
```

```
plaetten :: (a -> b -> c) -> (Baum a b c) -> [c]
```

# Kapitel 11.4

## *Ad hoc* Polymorphie auf Funktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

**11.4**

# Ad hoc Polymorphie

...eine schwächere, weniger generelle Form von Polymorphie mit folgenden Synonymen:

- ▶ Unechte Polymorphie.
- ▶ Überladen, Überladung (engl. Overloading).

## Definition 11.4.1 (*Ad hoc* polym. Fkt. in Haskell)

Eine Funktion in Haskell heißt *ad hoc polymorph* (oder *unecht polymorph* oder *überladen*), wenn die Typen eines oder mehrerer ihrer Parameter angegeben durch Typvariablen durch Typkontexte eingeschränkt Werte aller durch den Typkontext zugelassenen Typen als Argument zulassen.

Beispiele *ad hoc polymorpher* Funktionen:

```
(+)   :: Num a => a -> a -> a
(==)  :: Eq a  => a -> a -> Bool
(>)   :: Ord a => a -> a -> Bool
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Ad hoc Polymorphie auf Funktionen

...tritt in funktionalen wie nichtfunktionalen Sprachen ebenso **beiläufig** und **ubiquitär** auf wie **parametrische Polymorphie** in funktionalen Sprachen wie das Beispiel vieler vordefinierter überladener Funktionen zeigt:

- ▶ Die arithmetischen Funktionale  $(+)$ ,  $(*)$ ,  $(-)$ , etc.
- ▶ Die Booleschen Relatoren  $(==)$ ,  $(/=)$ ,  $(>)$ ,  $(>=)$ , etc.
- ▶ Die Zeige-Funktion **show** (Haskell-spezifisch)
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

901/170

# Wir unterscheiden genauer

...zwischen **direkt** und **indirekt überladenen** Funktionen:

- ▶ **(Direkt) überladene Funktionen**
  - ▶ **vordefinierter Typklassen:** Alle in einer vordefinierten Typklasse angegebenen Funktionen (z.B. `(==)`, `(/=)` aus `Eq`, `(<)`, `(>)` aus `Ord`, `(+)`, `(*)` aus `Num`, etc.)
  - ▶ **selbstdefinierter Typklassen:** Alle in einer selbstdefinierten Typklasse angegebenen Funktionen (z.B. `auswertung`, `reihenausw`, `gegläettet` aus `Analysierbar`; `warnung`, `warnreihe` aus `Warnung` (vgl. Kap. 4.3.5))
- ▶ **Indirekt überladene Funktionen:** Alle Funktionen, die sich auf eine überladene Funktion abstützen ohne selbst in einer Typklasse eingeführt zu sein, z.B.:

```
sum :: Num a => [a] -> a
sum []      = 0
sum (x:xs)  = x + sum xs
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

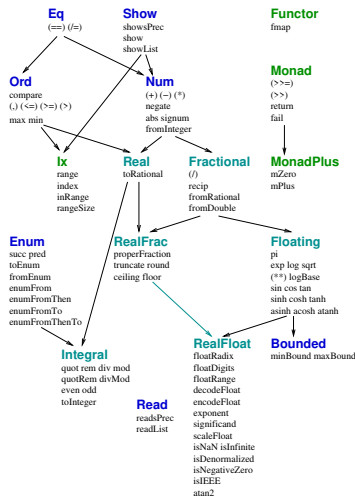
11.3

11.4

902/170

# Auswahl vor- und selbstdefinierter Typklassen

...zusammen mit den Namen der in ihnen eingeführten überladenen Funktionen:



Quelle: Fethi Rabhi, Guy Lapalme. *Algorithms - A Functional Approach*.  
Addison-Wesley, 1999, Abb. 2.4.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

903/1700

# Kapitel 11.4.1

## Überladene Funktionen vordefinierter Typklassen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

904/170



# Überladene Funktionen vordef. Typklassen

...am Beispiel der Typklassen (`Eq a`) und (`Num a`):

```
class Eq a where
  (==), (/=) :: a -> a -> Bool           -- Signaturen über-
                                          -- ladener Funktionen
                                          -- in Typklasse Eq

  x /= y = not (x==y)                    -- Protoimplemen-
  x == y = not (x/=y)                    -- tierungen

class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a           -- Signaturen über-
                                          -- ladener Funktionen
  negate :: a -> a                       -- in Typklasse Num
  abs, signum :: a -> a
  fromInteger :: Integer -> a

  x - y      = x + negate y              -- Protoimplemen-
  negate x   = 0 - x                     -- tierungen
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Weitere Bsp. überlad. Fkt. vordef. Typklassen

...die arithmetischen Funktionale und Funktionen:

(+) :: Num a => a -> a -> a

(\*) :: Num a => a -> a -> a

(/) :: Fractional a => a -> a -> a

div :: Integral a => a -> a

...die Booleschen Relatoren:

(==) :: Eq a => a -> a -> Bool

(/=) :: Eq a => a -> a -> Bool

(>) :: Ord a => a -> a -> Bool

(>=) :: Ord a => a -> a -> Bool

(<) :: Ord a => a -> a -> Bool

(<=) :: Ord a => a -> a -> Bool

...die Operatoren und Relatoren:

min :: Ord a => a -> a -> a

max :: Ord a => a -> a -> a

compare :: Ord a => a -> a -> Ordering

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

906/170

# Kapitel 11.4.2

## Überladene Funktionen selbstdefinierter Typklassen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Allgemeines Muster

...einer **Typklassendefinition** (vereinfacht):

```
class Name' tv => Name tv where
  f_1 :: ...           -- Signaturen der Typklassen-
  f_2 :: ...           -- funktionen f_1,...,f_k
  ...                  -- über tv und anderen Typen.
  f_k :: ...
  f_i = ...            -- Protoimplementierungen
  ...                  -- für null oder mehr der
  f_j = ...            -- Funktionen f_1,...,f_k.
```

Dabei:

- ▶ **Name'**: Name einer existierenden Typklasse als Kontext.
- ▶ **Name**: Freigewählter Name als Identifikator der Klasse.
- ▶ **tv**: Typvariable.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Überladene Funktionen in selbstdef. Typklassen

...am Beispiel der Typklassen `Info` und `Groesse`:

```
class Info a where
  wert_beispiele  :: [a]           -- Signaturen überla-
  zu_zeichenreihe :: a -> String  -- dener Funktionen
                                   -- in Typklasse Info

  wert_beispiele  = []             -- Protoimplemen-
  zu_zeichenreihe _ = ""           -- tierungen
                                   -- entspricht: zu_zeichenreihe = \_ -> ""

class Info a => Groesse a where
  groesse :: a -> Int              -- Signatur über-
                                   -- ladener Funktion
                                   -- in Typklasse Groesse

  groesse = (length . zu_zeichenreihe) -- Protoimple-
                                   -- mentierung
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Instanzbildungen für die Typen Char und Bool

...für die Typklassen Info und Groesse:

```
instance Info Char where
  wert_beispiele  = ['a','A','z','Z','0','9']
  zu_zeichenreihe = \c -> [c] (entspr.: zu_z. c = [c])

instance Groesse Char where
-- Die Protoimplementierung passte; nichts wäre zu
-- tun Aus Effizienzgründen geben wir dennoch an:
groesse = \_ -> 1          (entspr.: groesse _ = 1)

instance Info Bool where
  wert_beispiele      = [True,False]
  zu_zeichenreihe True  = "Wahr"
  zu_zeichenreihe False = "Falsch"

instance Groesse Bool where
  groesse True  = 4    -- length (zu_zeichenreihe True)
  groesse False = 6    -- length (zu_zeichenreihe False)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Instanzbildung für Typ Int

...für die Typklassen `Info` und `Groesse`:

```
instance Info Int where
  wert_beispiele      = [-42..42]
  zu_zeichenreihe n = < Code, der einen Int-Wert
                        durch seine Ziffernfolge in
                        Binärdarstellung als Zei-
                        chenreihe darstellt, z.B.
                        123  $\mapsto$  "1111011" >
```

```
instance Groesse Int
-- Die Protoimplementierung passt; nichts zu tun.
-- (Das Schlüsselwort where kann hier entfallen.)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

911/170

# Instanzbildung für Typ [a]

...für die Typklassen `Info` und `Groesse`:

```
instance Info a => Info [a] where
  wert_beispiele
    = [ [] ]
      ++ [ [x] | x <- wert_beispiele ]
      ++ [ [x,y] | x <- wert_beispiele,
                  y <- wert_beispiele ]
  zu_zeichenreihe = concat . (map zu_zeichenreihe)

instance Groesse a => Groesse [a] where
  groesse = ((foldr (+) 1) . (map groesse))
```

Beachte die überladene Verwendung der Funktionen:

- ▶ `wert_beispiele`, `zu_zeichenreihe`, `groesse` operieren auf Instanzen vom Typ `[a]`.
- ▶ `wert_beispiele`, `zu_zeichenreihe`, `groesse` operieren auf Instanzen vom Typ `a`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

912/170



# Instanzbildung für Typ (Baum a b c) (1)

...für die Typklassen `Info` und `Groesse`:

```
data Baum a b c = Blatt a b
                  | Wurzel (Baum a b c) c (Baum a b c)

instance (Info a, Info b, Info c) =>
    Info (Baum a b c) where

wert_beispiele
= [Blatt (head wert_beispiele) (last wert_beispiele),
   Wurzel
    (Blatt (wert_beispiele!!0) (wert_beispiele!!1))
    (wert_beispiele!!2)
    (Blatt (wert_beispiele!!1) (wert_beispiele!!0))]

zu_zeichenreihe baum
= < Code, der einen Baum-Wert als eine (i.a. mehrzei-
    lige) Zeichenreihe darstellt, die die Baumstruktur
    durch Einrückungen und Zeilenumbrüche deutlich
    macht. >
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Instanzbildung für Typ (Baum a b c) (2)

'Größe' kann vieles für einen Baum bedeuten, z.B.:

```
instance (Groesse a,Groesse b,Groesse c)
    => Groesse (Baum a b c) where
    groesse = (length . zu_zeichenreihe)
    -- d.h. Übernahme der Protoimplementierung.
```

...oder zweitens Größe als Tiefe des Baums:

```
instance (Groesse a,Groesse b,Groesse c)
    => Groesse (Baum a b c) where
    groesse = tiefe
```

...oder drittens Größe als Produkt bestimmter Baumparameter:

```
instance (Groesse a,Groesse b,Groesse c)
    => Groesse (Baum a b c) where
    groesse = ((2*) . length . gen_asstliste)
```

...oder viertens Größe als Summe von Blättern und Wurzeln,  
oder sechstens als Anzahl der {a,b,c}-Marken oder siebentes...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Das Hugs-Kommandos :t

...hilft, auf einfache Art den **allgemeinsten Typ** eines gültigen **Haskell**-Ausdrucks zu bestimmen, z.B.:

```
Main> :t wert_beispiele  
wert_beispiele :: Info a => [a]
```

```
Main> :t zu_zeichenreihe  
zu_zeichenreihe :: Info a => a -> [Char]
```

```
Main> :t groesse  
groesse :: Groesse a => a -> Int
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Anwendungsbeispiele (1)

...der überladenen Funktionen der Typklassen `Info` und `Groesse`:

```
wert_beispiele :: Bool ->> [True,False]
wert_beispiele :: Char ->> ['a','A','z','Z','0','9']
[first (wert_beispiele :: Int)]
  ++ [last (wert_beispiele :: Int)]    ->> [-42,42]
([first wert_beispiele] :: [Int])
  ++ ([last wert_beispiele] :: [Int]) ->> [-42,42]

zu_zeichenreihe True ->> "Wahr"
zu_zeichenreihe '5'   ->> "5"
zu_zeichenreihe 123   ->> "1111011"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

916/170

# Anwendungsbeispiele (2)

Abkürzungen: `cc` für `concat`, `zz` für `zu_zeichenreihe`.

```
zu_zeichenreihe [1,2,3]
```

```
->> zz [1,2,3]
```

```
->> (cc . (map zz)) [1,2,3]
```

```
->> cc (map zz [1,2,3])
```

```
->> cc [zz 1,zz 2,zz 3]
```

```
->> cc ["1","10","11"]
```

```
->> cc [['1'], ['1','0'], ['1','1']]
```

```
->> ['1','1','0','1','1']
```

```
->> "11011"
```

```
zu_zeichenreihe [[1,2,3],[4,5],[6]]
```

```
->> zz [[1,2,3],[4,5],[6]]
```

```
->> (cc . (map zz)) [[1,2,3],[4,5],[6]]
```

```
->> cc (map zz [[1,2,3],[4,5],[6]])
```

```
->> cc [zz [1,2,3],zz [4,5],zz [6]]
```

```
->> ...
```

```
->> cc ["11011","100101","110"]
```

```
->> ...
```

```
->> "11011100101110"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

917/170

# Anwendungsbeispiele (3)

...mit ergänzten Zwischenschritten.

```
zu_zeichenreihe [[1,2,3],[4,5],[6]]
->> zz [[1,2,3],[4,5],[6]]
->> (cc . (map zz)) [[1,2,3],[4,5],[6]]
->> cc (map zz [[1,2,3],[4,5],[6]])
->> cc [zz [1,2,3],zz [4,5],zz [6]]
->> cc [(cc . (map zz)) [1,2,3],(cc . (map zz)) [4,5],
      (cc . (map zz)) [6]]
->> cc [cc (map zz [1,2,3]),cc (map zz [4,5]),
      cc (map zz [6])]
->> cc [cc [zz 1,zz 2,zz 3],cc [zz 4,zz 5],cc [zz 6]]
->> cc [cc ["1","10","11"],cc ["100","101"],cc ["110"]]
->> cc [cc [['1'],['1','0'],['1','1']],cc [['1','0','0'],['1','0','1']],
      cc [['1','1','0']]]
->> cc [['1','1','0','1','1'],['1','0','0','1','0','1'],['1','1','0']]
->> ['1','1','0','1','1','1','0','0','1','0','1','1','1','0']
->> "11011100101110"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Anwendungsbeispiele (4)

```
groesse False ->> 6
groesse 'z'    ->> 1
groesse 123
->> (length . zz) 123
->> length (zz 123)
->> length "1111011"
->> 7
groesse [[1,2,3],[4,5],[6]]
->> foldr (+) 1 (map (length . zz) [[1,2,3],[4,5],[6]])
->> foldr (+) 1 [(length . zz) [1,2,3],(length . zz) [4,5],
               (length . zz) [6]]
->> foldr (+) 1 [length (zz [1,2,3]),length (zz [4,5]),
               length (zz [6])]
->> foldr (+) 1 [length "11011",length "100101",length "110"]
->> foldr (+) 1 [5,6,3]
->> 1 + foldr (+) 0 [5,6,3]
->> 1 + 14
->> 15
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Übungsaufgabe 11.4.2.1

Ergänze **fehlende Codestücke** in den Beispielen:

- ▶ Instanzbildung (**Info Int**): Vervollständige die Implementierung der Funktion **zu\_zeichenreihe**.
- ▶ Instanzbildung (**Info (Baum a b c)**): Vervollständige die Implementierung der Funktion **zu\_zeichenreihe**.
- ▶ Instanzbildung (**Groesse (Baum a b c)**): Vervollständige die Implementierung d. Funktion **groesse** mit 'Größe' als
  - ▶ Summe von Blättern und Wurzeln,
  - ▶ Anzahl der **{a,b,c}**-Marken.

Überlege mindestens eine weitere Variante, die als 'Größe' von Bäumen verstanden werden kann und nimm die zugehörige(n) Instanzbildung(en) für diese Variante(n) vor.

Teste die **Implementierungen** anhand geeigneter Beispiele, ob sie das gewünschte Verhalten zeigen; ebenso die in diesem Kapitel angegebenen **Instanzbildungsimpementierungen**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4



# Übungsaufgabe 11.4.2.2

Führe die **Schritt- für Schrittauswertung** der überladenen Funktionen **wert\_beispiele**, **zu\_zeichenreihe** und **groesse** der Typklassen **Info** und **Groesse** für weitere Werte aus, z.B. für Werte der Typen:

- ▶ **String**, z.B. für die Werte **"** und **"abcd"**.
- ▶ **[Bool]**, z.B. für die Werte **[]** und **[True,False,True]**.
- ▶ **[[[Int]]]**, z.B. für die Werte **[]**,  **[[]]**, **[[[]]]** und **[[[1,2,3],[4,5]],[[6],[7,8]]]**.
- ▶ **(Baum Int Int Int)**, z.B. für die Werte **(Blatt 17 4)** und **(Wurzel (Blatt 1 2) 3 (Blatt 4 5))**.
- ▶ ...

# Übungsaufgabe 11.4.2.3

Mache weitere **Typen** zu **Instanzen** der Typklassen **Info** und **Groesse**, z.B. die Typen:

- ▶ **Float** (s. Kap. 2.1.3)
- ▶ **Pegelstand** (s. Kap. 4.2.2)
- ▶ **Mensch** (s. Kap. 5.1)
- ▶ ...

Teste die **Implementierungen** auf gewünschtes Verhalten und führe auch hier für jeweils einige Datenbeispiele **Schritt-für-Schritt-Auswertungen** der überladenen Funktionen **wert\_beispiele**, **zu\_zeichenreihe** und **groesse** durch.

# Kapitel 11.4.3

## Vererben, erben, überschreiben

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Vererben, erben, überschreiben

Typklassen können

- ▶ Spezifikationen **mehr als einer** Funktion bereitstellen.
- ▶ **Protoimplementierungen** (engl. **default implementations**) für (alle oder einige) dieser Funktionen **bereitstellen**.
- ▶ von anderen Typklassen **erben**.
- ▶ geerbte Implementierungen **überschreiben**.

In der Folge betrachten wir dies anhand einiger Beispiele in **Haskell** vordefinierter Typklassen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Vererben, erben und überschreiben

...auf Typklassenebene:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x==y) -- Protoimplementierung f. (/=)
  x == y = not (x/=y) -- Protoimplementierung f. (==)

class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min             :: a -> a -> a
  compare              :: a -> a -> Ordering
  x <= y = (x < y) || (x == y) -- Protoimpl. f. (<=)
  x > y  = y < x               -- Protoimpl. f. (<)
  ...
  compare x y -- Protoimplementierung f. compare
    | x == y   = EQ
    | x <= y   = LT
    | otherwise = GT
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Anmerkungen

- ▶ Die Typklasse `Ord` erweitert die Klasse `Eq`; jeder Typ, der zu einer Instanz der Typklasse `Ord` gemacht werden soll, muss bereits Instanz der Typklasse `Eq` sein.
- ▶ Jede Typinstanz von `Ord` **erbt** die Implementierungen ihrer Instanz aus `Eq`; umgekehrt **vererbt** jede Typinstanz aus `Eq` ihre Implementierungen an ihre Instanzen aus `Ord`.
- ▶ Für jede Typinstanz `T` der Typklasse `Ord` darf man sich darauf verlassen, dass es für `T`-Werte Implementierungen für alle Funktionen der Typklassen `Eq` und `Ord` gibt.
- ▶ Die Typklassen `Eq` und `Ord` stellen für einige Funktionen bereits Protoimplementierungen bereit.
- ▶ Für eine vollständige Instanzbildung reicht es deshalb, Implementierungen der Relatoren `(==)` und `(<)` anzugeben.
- ▶ Leisten die Protoimplementierungen nicht das Gewünschte, können sie durch instanzspezifische Implementierungen **überschrieben** werden.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Überschreiben automatisch generierter Impl.

...am Beispiel der `Eq`-Instanzbildung für den Typ `(Paar a)`:

```
newtype Paar a = P (a,a)
```

```
instance (Eq a) => Eq (Paar a) where
```

```
  P (u,v) == P (x,y) = (u == x) && (v == y)
```

- **Automatisch generiert:** Die `Eq`-Instanz `(Paar a)` erhält für `(/=)` folgende sich aus den Protoimplementierungen der Klasse automatisch ergebende Implementierung:

```
P x /= P y = not (P x == P y)
```

- **Überschreiben:** Die sich automatisch ergebende Implementierung von `(/=)` kann bei der Instanzbildung für `(Paar a)` überschrieben werden, z.B. durch folgende (geringfügig) effizientere Fassung:

```
instance (Eq a) => Eq (Paar a) where
```

```
  P (u,v) == P (x,y) = (u == x) && (v == y)
```

```
  P (u,v) /= P (x,y) = if u /= x then True else v /= y
```

# Mehrfachvererben, -erben und -überschreiben

...auf Typklassenebene ist ebenfalls möglich; Haskell's vordefinierte Typklasse `Num` ist ein Beispiel dafür:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate       :: a -> a
  abs, signum  :: a -> a
  fromInteger  :: Integer -> a   -- Zwei Typkonver-
  fromInt      :: Int -> a       -- sionsfunktionen!

  x - y        = x + negate y    -- Protoimpl.
  fromInt      = ...
```

...jede Instanz der Typklasse `Num` muss bereits zum Zeitpunkt der Instanzbildung Instanz der Typklassen `Eq` und `Show` sein.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4



# Übungsaufgabe 11.4.3.1

Vergleiche das Vererbungskonzept von Haskell mit dem Vererbungskonzept objektorientierter Sprachen, z.B. von Java.

Welche Gemeinsamkeiten, welche Unterschiede gibt es?

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Kapitel 11.4.4

## Automatische Typklasseninstanzbildung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Automatische Typklasseninstanziierung

...möglich für bestimmte vordefinierte Typklassen mithilfe der `deriving`-Klausel:

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
                deriving (Eq,Ord,Enum,Bounded,
                        Show,Read)

data Suchbaum = Leer | Knoten Suchbaum Integer Suchbaum
               deriving (Eq,Ord,Show,Read)

newtype GanzeZahlen = GZ Int deriving (Eq,Ord,Bounded,
                                       Show,Read)

data (Ord a, Ord b, Ord c) => Baum a b c
  = Blatt a b
  | Wurzel (Baum a b c) c (Baum a b c)
            deriving (Eq,Ord,Show)

newtype (Ord a, Ord b, Show a, Show b) =>
  Relation a b = R [(a,b)] deriving (Eq,Ord,Show)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Algebraische und neue Typen

...können mithilfe einer

- ▶ **deriving**-Klausel **automatisch** als Instanzen (einer festen Auswahl) vordefinierter Typklassen angelegt werden (keine Typsynonyme!).

Für die Funktionen der in der **deriving**-Klausel angeführten Typklassen wird dabei das

- ▶ **'Offensichtliche'** als Standardimplementierung generiert.

Intuitiv ersetzt die Angabe einer **deriving**-Klausel mit einer oder mehreren Typklassen

- ▶ die Angabe der entsprechenden **instance**-Deklaration.
- ▶ Elementare Typen (**Int**, **Float**, **Bool**, **Char**, etc.), Zeichenreihen (**String**) und Tupel und Listen solcher Typen sind vordefinierte Instanzen der infrage kommenden Typklassen (**Integer** z.B. nicht für die Typklasse **Bounded**).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Autom. vs. manuelle Typklasseninstanziierung

Beispiel: Die Deklaration mit `deriving`-Klausel:

```
data (Eq a, Eq b, Eq c) => Baum a b c
    = Blatt a b
    | Wurzel (Baum a b c) c (Baum a b c) deriving Eq
```

ist gleichbedeutend zum Deklarationspaar:

```
data (Eq a, Eq b, Eq c) => Baum a b c
    = Blatt a b
    | Wurzel (Baum a b c) c (Baum a b c)

instance (Eq a, Eq b, Eq c) => Eq (Baum a b c) where
    (Blatt u v) == (Blatt x y) = (u == x) && (v == y)
    (Wurzel ltb z rtb) == (Wurzel ltb' z' rtb')
        = (ltb == ltb') && (z == z') && (rtb == rtb')
    _ == _ = False
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Autom. vs. manuelle Typklasseninstanzbildung

...bzw. zum Deklarationspaar:

```
data Baum a b c = Blatt a b
                  | Wurzel (Baum a b c) c (Baum a b c)

instance (Eq a, Eq b, Eq c) => Eq (Baum a b c) where
  (Blatt u v) == (Blatt x y) = (u == x) && (v == y)
  (Wurzel ltb z rtb) == (Wurzel ltb' z' rtb')
    = (ltb == ltb') && (z == z') && (rtb == rtb')
  _ == _ = False
```

...mit 'offensichtlicher' Gleichheit interpretiert als Gleichheit in Struktur und Benennung.

Beachte: Der Kontext '(Eq a, Eq b, Eq c) =>' in den Instanzdeklarationen ist nötig für die Instanzbildungen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

11.5

11.6

11.7

934/170

# Flexibilität durch manuelle Instanzbildung (1)

Manuelle Typklasseninstanzbildung erlaubt Gleichheit abweichend von 'offensichtlicher' Gleichheit in (nahezu) jeder gewünschten Weise zu implementieren:

```
data Baum a b c = Blatt a b
                  | Wurzel (Baum a b c) c (Baum a b c)
```

Z.B. als Gleichheit in Struktur und  $\{a,c\}$ -Benennungen, mit unbeachtet bleibenden (und möglicherweise unterschiedlichen)  $b$ -Benennungen:

```
instance (Eq a, Eq c) => Eq (Baum a b c) where
  (Blatt u _) == (Blatt x _) = (u == x)
  (Wurzel ltb z rtb) == (Wurzel ltb' z' rtb')
    = (ltb == ltb') && (z == z') && (rtb == rtb')
  _ == _ = False
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Flexibilität durch manuelle Instanzbildung (2)

...oder als Gleichheit der Benennungen in mengenartigem Sinn:

```
instance (Ord a, Ord b, Ord c) => Eq (Baum a b c) where
  b == b' = (kollabiere b) == (kollabiere b')
```

wobei

```
kollabiere :: (Ord a, Ord b, Ord c) =>
              (Baum a b c) -> ([a],[b],[c])
kollabiere = (entferne_duplikate . sortiere . aufsammeln)

aufsammeln :: (Baum a b c) -> ([a],[b],[c])
aufsammeln (Blatt x y) = ([x],[y],[ ])
aufsammeln (Wurzel ltb z rtb)
  = (aufsammeln ltb) +++ ([],[ ],[z]) +++ (aufsammeln rtb)

(+++) :: ([a],[b],[c]) -> ([a],[b],[c]) -> ([a],[b],[c])
(xs,ys,zs) +++ (xs',ys',zs') = (xs++xs',ys++ys',zs++zs')
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4



# Flexibilität durch manuelle Instanzbildung (3)

...und:

```
sortiere :: (Ord a, Ord b, Ord c)
          => ([a],[b],[c]) -> ([a],[b],[c])
```

```
sortiere (xs,ys,zs)
  = (quickSort xs,quickSort ys,quickSort zs)
```

```
entferne_duplikate :: (Eq a, Eq b, Eq c)
                   => ([a],[b],[c]) -> ([a],[b],[c])
```

```
entferne_duplikate (xs,ys,zs)
  = < Code zum Entfernen von Duplikaten von Elementen >
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Automatische Typklasseninstanzbildung

...ist möglich (ausschließlich!) für folgende Menge **vordefinierter Typklassen** in **Haskell**:

- ▶ **Eq**
- ▶ **Ord**
- ▶ **Enum**
- ▶ **Bounded**
- ▶ **Show**
- ▶ **Read**

Für andere Typklassen, gleich ob vor- oder selbstdefiniert, sind zur Instanzbildung stets **instance**-Deklarationen erforderlich; das gilt ebenso, falls von 'offensichtlicher' abweichende Bedeutungen einer oder mehrerer Typklassenfunktionen gewünscht sind.

# Übungsaufgabe 11.4.4.1

Ergänze den Code für die Funktion `entferne_duplikate` und teste die verschiedenen `Eq`-Instantiierungsvarianten für den Datentyp `(Baum a b c)`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Kapitel 11.4.5

## Grenzen des Überladens

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Ist es möglich

...jeden Typ zu einer Instanz der Typklasse `Eq` zu machen?

*De facto* hieße das, den Typ des Gleichheitsrelators `(==)` von

`(==) :: Eq a => a -> a -> Bool`

über das Mittel des Überladens auf

`(==) :: a -> a -> Bool`

zu verallgemeinern; genauer, so nahe wie immer gewünscht daran anzunähern?

# Im Sinne von

...Funktionen als **erstrangigen Sprachelementen** (engl. **first class citizens**) wäre ein Gleichheitstest auf Funktionen höchst wünschenswert, z.B.

```
(==) fac fib                ->> False
(==) (\x -> x+x) (\x -> 2*x) ->> True
(==) (+2) (2+)              ->> True
```

Anders als z.B. für die Parametervertauschung durch das Funktional

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

...ist **Gleichheit** eine typabhängige Eigenschaft, die eine **typspezifische** Implementierung erfordert.

# In Haskell

...erforderte dies `Eq`-Instanzbildungen auch für funktionale Typen vorzunehmen, etwa für die Typen `(Int -> Int)` und `(Int -> Int -> Int)` für die Abdeckung der Beispiele:

```
instance Eq (Int -> Int) where  
  (==) f g = ...
```

```
instance Eq (Int -> Int -> Int) where  
  (==) f g = ...
```

**Preisfrage:** Können wir die 'Punkte' so ersetzen, dass wir eine valide Gleichheitsprüfung für alle Paare von Funktionen der Typen `(Int -> Int)` und `(Int -> Int -> Int)` erhalten?

**Antwort: Nein!**

# Gleichheit von Funktionen: Unentscheidbar

## Theorem 11.4.5.1 (Theoretische Informatik)

Gleichheit von Funktionen ist nicht entscheidbar, d.h. es gibt keinen Algorithmus, der für zwei beliebig vorgelegte Funktionen stets nach endlich vielen Schritten entscheidet, ob diese Funktionen gleich sind oder nicht.

Wichtig: Theorem 11.4.5.1 schließt nicht aus, dass für konkret vorgelegte Funktionen oder bestimmte Klassen von Funktionen deren Gleichheit (algorithmisch) entschieden werden kann.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4



# Abwesenheit universeller, allumfassender Lsg.

...ein typisches Problem.

Siehe [Moshe Vardis](#) Kolumne

- ▶ Moshe Vardi. [Self-Reference and Section 230](#). Communications of the ACM 61(11):7, 2018.

zu

- ▶ [Konsistenz](#) (Widerspruchsfreiheit, Freiheit von Paradoxien)
- ▶ [Vollständigkeit](#) (Alles ist beantwortbar)
- ▶ [Entscheidbarkeit](#) (Berechenbarkeit, Automatisierbarkeit)

formaler und nichtformaler Systeme von [Mathematik](#) und [Logik](#) über [Informatik](#) zu [Gesellschaftstheorie](#), von [Eubulides](#) ([Paradoxon des Lügners](#)) über [Frege](#) ([Axiomatisierung der Mengentheorie](#)) und [Hilbert](#) ([Hilberts Programm](#)) zu [Russel](#) ([Mengenparadoxon](#)), [Gödel](#) ([Unvollständigkeit der Arithmetik](#), [Konsistenz der Arithmetik nicht innerhalb der Arithmetik beweisbar](#)), [Church](#) und [Turing](#) ([Unentscheidbarkeit der Prädikatenlogik](#)) zu [Popper](#) ([Freiheits- und Toleranzparadoxon](#)).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

945/170

# Zusammenfassend

...anhand der Beobachtungen am Gleichheitsrelator:

- ▶ Funktionen bestimmter (auch scheinbar universeller Natur) Funktionalität lassen sich i.a. nicht für jeden Typ angeben, sondern nur für eine Teilmenge aller Typen.
- ▶ Die Funktionalität des Gleichheitsrelators ist ein konkretes Beispiel einer solchen Funktion.
- ▶ Auch wenn es verlockend wäre, für alle Typen den
  - ▶ Gleichheitsrelator **echt** oder zumindest **parametrisch polymorph** implementieren zu können, in **Haskell** mit der Signatur: `(==) :: Eq a => a -> a -> Bool`ist dies in dieser Form und Allgemeinheit in **keiner (!)** Sprache möglich!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# In Haskell

...sind die Typen, auf deren Werten der

- Gleichheitsrelator (`==`) definiert ist

genau die **Elemente** (oder **Instanzen**) der Typklasse `Eq`.

Bei der `Eq`-Instanzbildung für einen Typ `T` (gleich ob manuell oder automatisch) wird die

- **typspezifische Bedeutung** des Gleichheitsrelators für `T`-Werte

durch explizite Ausprogrammierung von Gleichheits- und Ungleichheitsrelator (`==`) und (`/=`) definiert und exakt festlegt.

# Kapitel 11.4.6

## *Ad hoc* Polymorphie vs. Polymorphie

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

948/170

# Ad hoc Polymorphie über Typklassen

Typklassen sind

- **Kollektionen** von Typen, auf deren Werten die in der Typklasse angegebenen Funktionen definiert sind.

Durch **Instanzbildungen** der Typklasse für verschiedene Typen wird die Bedeutung

- dieser Funktionen **überladen** und **typspezifisch**.

**Zweckmäßiger** (wie auch **üblicherweise**) **Weise** sind

- die typspezifischen Bedeutungen der überladenen Funktionen einander **'entsprechend'**, ihre Funktionalität einander **'vergleichbar'**, jeweils typspezifisch zugeschnitten.
- **Instanzbildung mit 'vergleichbarer' Funktionalität** kann nicht syntaktisch erzwungen werden; sie liegt in der Verantwortung des Programmierers und richtet einen Appell an die **Programmierdisziplin**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Wiederverwendung durch *ad hoc* Polymorphie

*Ad hoc* Polymorphie (oder *unechte* Polymorphie oder *Überladung*) unterstützt *Wiederverwendung* des

- Funktionsnamen, nicht jedoch der Funktionsimplementierung (diese wird typspezifisch bei der Typklasseninstanziierung ausprogrammiert, ggf. unter Ausnutzung der Protoimplementierungen der jeweiligen Typklasse):

```
(+)           :: Num a      => a -> a -> a
(>)           :: Ord a      => a -> a -> Bool
zu_zeichenreihe :: Info a  => a -> String
groesse       :: Groesse a => a -> Int
```

d.h. es gilt das Prinzip:

- ein Name, eine T-spezifische Implementierung pro Instanz T von a.

# Wiederverw. durch parametrische Polymorphie

Parametrische (oder echte) Polymorphie unterstützt Wiederverwendung von

- Funktionsname und -implementierung:

```
curry :: ((a,b) -> c) -> a -> b -> c
```

```
curry f x y = f (x,y)
```

```
length :: [a] -> Int
```

```
length [] = 0
```

```
length (x:xs) = 1 + length xs
```

```
fst :: (a,b) -> a
```

```
fst (x,_) = x
```

d.h. es gilt das Prinzip:

- ein Name, eine Implementierung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Ad hoc Polymorphie vs. Polymorphie (1)

**Polymorphie:** Ein **polymorpher Typ** wie  $(a \rightarrow a)$  steht informell für:

$$\forall a \in \text{"Menge gültiger Typen"}. (a \rightarrow a)$$

- Funktionen wie  $\text{id} :: a \rightarrow a$  besitzen genau eine Implementierung und sind für jeden gültigen Haskell-Typ eine Funktion vom Typ  $(a \rightarrow a)$ .

**Ad hoc Polymorphie:** Ein **ad hoc polymorpher Typ** wie  $(\text{Num } a \Rightarrow a \rightarrow a \rightarrow a)$  steht informell für:

$$\forall a \in \text{Num}. (a \rightarrow a \rightarrow a)$$

- Funktionen wie  $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$  besitzen für jede **Instanz der Typklasse Num** eine instanzspezifische Implementierung und sind für jeden dieser Instanztypen eine Funktion vom Typ  $(a \rightarrow a \rightarrow a)$ ; für sonst keinen.



# Ad hoc Polymorphie vs. Polymorphie (2)

Parametrische Polymorphie ist unter dem Aspekt Wiederverwendung echt stärker als *ad hoc* Polymorphie.

**Dennoch:** Bereits die von *ad hoc* Polymorphie unterstützte Wiederverwendung von Funktionsnamen ist **äußerst nützlich**.

Ohne *ad hoc* Polymorphie wären **typspezifische Namen** erforderlich nicht nur für

- **eigendefinierte Funktionen** ( $\text{groesse}_{Int}$ ,  $\text{groesse}_{Bool}$ ,  $\text{groesse}_{Char}$ , etc.)

sondern auch für bekannte Standardoperatoren wie

- **Boolesche Relatoren** ( $=_{Bool}$ ,  $>_{Float}$ ,  $<_{String}$ , etc.) und **arithmetische Operatoren** ( $+_{Int}$ ,  $-_{Float}$ ,  $*_{Double}$ , etc.)

Deren zwingender Gebrauch wäre nicht nur ungewohnt und unschön, sondern auch äußerst lästig im täglichen Gebrauch.

**Anmerkung:** Andere Sprachen wie ML und Opal gehen hier einen anderen Umsetzungsweg als Haskell und bieten andere Konzepte als Typklassen.

# Kapitel 11.5

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Polymorphie

...auf Funktionen und Datentypen unterstützt

- Wiederverwendung durch Parametrisierung

und damit die

- Ökonomie der Programmierung (flapsig: *Schreibfaulheit*).

durch Ausnutzung der Beobachtung, dass **tragende Eigenschaften** eines Datentyps wie von darauf arbeitenden Funktionen oft **unabhängig** von typspezifischen Details sind.

Insgesamt: Ein **typisches Vorgehen** in der **Informatik**:

- 'Gleiche' Teile werden 'ausgeklammert' und dadurch einer Wiederverwendung zugänglich gemacht.
- Im Fall von **Polymorphie** bedeutet das, dass ansonsten i.w. gleiche Codeteile **nicht** mehrfach geschrieben werden müssen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Als Gratis-Nebeneffekt

...trägt **Polymorphie** bei zu **höherer**

- ▶ **Transparenz und Lesbarkeit**

...durch Betonung von Gemeinsamkeiten, nicht von Unterschieden.

- ▶ **Verlässlichkeit und Wartbarkeit**

...hinsichtlich Fehlersuche, Weiterentwicklung, etc.

- ▶ **Programmiereffizienz**

...hinsichtlich höherer Produktivität, früherer Markteintrittsmöglichkeit (engl. time-to-market).

- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Nichtzuletzt: Polymorphie

...ein aktuelles **Forschungs-** und **Entwicklungsgebiet** auch in anderen **Paradigmen**, speziell dem

► **objektorientierter** Programmierung.

Nutzen und Vorteile **polymorpher** Konzepte für Datentypen und Funktionen werden zunehmend auch für Datentypen und Methoden zu schätzen gewusst (Stich- und Schlagwort: **Generic Java**).

Res amicos invenit.  
Die Sache findet Freunde.

Persius (34 - 62 n.Chr.)  
röm. Dichter

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Abschliessend

...über Teil IV 'Funktionale Programmierung' der Vorlesung:

Die Stärken des funktionalen Programmierstils resultieren aus insgesamt wenigen Konzepten für

- ▶ Funktionen
- ▶ Datentypen

Tragend sind dabei die Konzepte von

- ▶ Funktionen als erstrangige Sprachelemente (engl. first class citizens)
  - ▶ Stichwort: Funktionen höherer Ordnung (Kap. 10)
- ▶ Polymorphie als durchgängiges Prinzip auf
  - ▶ Datentypen (Kap. 11.2)
  - ▶ Funktionen (Kap. 11.3, Kap. 11.4)

# Wenige Faktoren, mächtiges Zusammenspiel

Kombination und nahtloses Zusammenspiel der tragenden (wenigen) Einzelkonzepte führen in Summe zur hohen

► Ausdruckskraft und Flexibilität

des funktionalen Programmierstils.

Das Ganze ist mehr als die Summe seiner Teile.

Aristoteles (384 - 322 v.Chr.)  
griech. Philosoph

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Paradigmen- u. sprachspez. Automatismen

Speziell in **Haskell** tragen zu Ausdruckskraft und Flexibilität weitere paradigm- und sprachspezifische **Automatismen** bei, etwa zur **automatischen Generierung** von:

- ▶ **Listen**: `[2,4..42]`, `[odd n | n <- [1..], n<1000]`.
- ▶ **Selektorfunktionen**: Verbundtyp-Syntax für algebraische Datentypen.
- ▶ **Typklasseninstanzen**: **deriving**-Klausel.

Siehe

- ▶ Peter Pepper. **Funktionale Programmierung in OPAL, ML, Haskell und Gofer**. Springer-V., 2. Auflage, 2003.

für eine weiterführende und vertiefende Diskussion.



# Kapitel 11.6

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

11.1

11.2

11.3

11.4

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 11 (1)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 7, Eigene Typen und Typklassen definieren)
-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 5, Polymorphic and Higher-Order Functions; Kapitel 9, More about Higher-Order Functions; Kapitel 12, Qualified Types; Kapitel 24, A Tour of Haskell's Standard Type Classes)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3, Types and classes; Kapitel 8, Declaring types and classes)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11




11.1

11.2

11.3

11.4

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 11 (2)

-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 2, Believe the Type; Kapitel 7, Making our own Types and Type Classes)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 19, Formalismen 4: Parametrisierung und Polymorphie)
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 2.8, Type classes and class methods)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11




11.1

11.2

11.3

11.4

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 11 (3)

-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999.  
(Kapitel 12, Overloading and type classes; Kapitel 14.3, Polymorphic algebraic types; Kapitel 14.6, Algebraic types and type classes)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.  
(Kapitel 13, Overloading, type classes and type checking; Kapitel 14.3, Polymorphic algebraic types; Kapitel 14.6, Algebraic types and type classes)
-  Moshe Vardi. *Self-Reference and Section 230*. Communications of the ACM 61(11):7, 2018.

# Teil V

## Fundierung funktionaler Programmierung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

**Teil V**

Kap. 12

Kap. 13

# Kapitel 12

## $\lambda$ -Kalkül

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

**Kap. 12**

# Kapitel 12.1

## Motivation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

*“...much of our attention is focused on functional programming, which is the most successful programming paradigm founded on a rigorous mathematical discipline. Its foundation, the lambda calculus, has an elegant computational theory and is arguably the smallest universal programming language. As such, the lambda calculus is also crucial to understand the properties of language paradigms other [than] functional programming...”*

Exzerpt von der Startseite der  
‘Programming Languages and Systems (PLS)’  
Forschungsgruppe an der University of New South Wales,  
Sydney, geleitet von Manuel Chakravarty und Gabriele Keller.  
( <http://www.cse.unsw.edu.au/~pls/PLS/PLS.html> )



Der  $\lambda$ -Kalkül (mit  $\lambda$ -definierbaren Funktionen) ist

- ▶ formales Berechenbarkeitsmodell neben anderen, u.a.:
  - ▶ Turing-Maschinen
  - ▶ Markov-Algorithmen
  - ▶ Allgemein rekursive Funktionen
  - ▶ ...
- ▶ fundamental für die Berechenbarkeitstheorie.
- ▶ formale Fundierung funktionaler Programmierung und Programmiersprachen.

# Berechenbarkeitstheorie

...im **Mittelpunkt** stehende Fragen:

- ▶ Was **heißt** berechenbar?
- ▶ Was **ist** berechenbar?
- ▶ Wie **aufwändig** ist etwas zu berechnen?
- ▶ Gibt es **Grenzen** der Berechenbarkeit?
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Intuitive Berechenbarkeit

...ein informeller Berechenbarkeitsbegriff.

‘Etwas’ ist intuitiv berechenbar, wenn es

- ▶ eine irgendwie machbare effektive mechanische Methode gibt, die zu jedem gültigen Argumentwert in endlich vielen Schritten den Funktionswert konstruiert und für alle anderen Argumentwerte entweder mit einem speziellen Fehlerwert oder nie abbricht.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Intuitive Berechenbarkeit, intuitiv berechenbar

...was ist damit gewonnen?

Für die Beantwortung der sehr konkreten Fragen der **Berechenbarkeitstheorie** zunächst einmal **nichts**, da die Bedeutung von **intuitiv berechenbar**

- ▶ vollkommen **vage** und **nicht greifbar** ist, ein **Bauchgefühl**:

*“...wenn es eine **irgendwie machbare** effektive mechanische Methode gibt...”*

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Formale Berechenbarkeit, formal berechenbar

Grundlegende Aufgabe der **Berechenbarkeitstheorie**:

- ▶ **Berechenbarkeitsbegriffe** zu ersinnen und so zu konkretisieren, dass sie
  - ▶ **formal** gefasst
  - ▶ einer **präzisen Behandlung** zugänglich gemacht
  - ▶ bezüglich ihrer Ausdruckskraft und Stärke miteinander **verglichen**werden können.

Zentral: Die **Einführung**

- ▶ **formaler Berechnungsmodelle**

die den Begriff 'berechenbar' **präzise** und **rigoros** definieren und damit

- ▶ handfeste **Ausprägungen** von **Berechenbarkeit** liefern.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Wichtige Bsp. formaler Berechnungsmodelle

...und ihre zeitliche Einordnung:

- ▶ Allgemein rekursive Funktionen (Herbrand 1931, Gödel 1934, Kleene 1936) (s. Anh. B.3, B.4)
- ▶  $\mu$ -rekursive Funktionen (Kleene 1936) (s. Anh. B.4)
- ▶ Turing-Maschinen (Turing 1936) (s. Anh. B.1)
- ▶ Endliche kombinatorische Prozesse (Post 1936)
- ▶ Markov-Algorithmen (Markov 1951) (s. Anh. B.2)
- ▶ Registermaschinen (Random Access Machines (RAMs)) (Shepherdson, Sturgis 1963)
- ▶ ...
- ▶  $\lambda$ -definierbare Funktionen,  $\lambda$ -Kalkül (Church 1936)

# Holzschnittartige Charakterisierung

...formaler Berechnungsmodelle:

Maschinenbasierte/-orientierte Konkretisierungen von Berechenbarkeit:

- ▶ Turing-Maschinen, Registermaschinen.

Programmierbasierte/-orientierte Konkretisierungen von Berechenbarkeit:

- ▶ Markov-Algorithmen, Theorie rekursiver Funktionen,  $\lambda$ -Kalkül.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Intuitive Berechenbarkeit u. Churchsche These

## Die Churchsche These

Eine Funktion ist **intuitiv berechenbar** genau dann, wenn sie  **$\lambda$ -definierbar** ist, im  **$\lambda$ -Kalkül** (als Funktion über den natürlichen Zahlen) ausdrückbar ist.

Ein **Beweis** für diese These?

- ▶ Rückrichtung: Trivial.
- ▶ Hinrichtung: Unmöglich.

...aufgrund der **Schwammigkeit** und daraus folgenden **Nicht-fassbarkeit** des Begriffs 'intuitiv berechenbar' entzieht sich die Hinrichtung der **Churchschen These** jedem Beweisversuch.

Zur **Church/Turingschen These** siehe z.B.:

B. Jack Copeland. **The Church-Turing Thesis**. The Stanford Encyclopedia of Philosophy, 2002.

<http://plato.stanford.edu/entries/church-turing>

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12



# Gleichmächtigkeit von Berechnungsmodellen

...man hat jedoch folgendes **beweisen** können:

## Theorem 12.1.1 (Gleichmächtigkeit)

Alle der eingangs genannten **formalen Berechnungsmodelle** sind

► **gleich mächtig**

d.h. was in einem Modell berechenbar ist, ist in jedem der anderen Modelle berechenbar und umgekehrt.

Dieses Resultat kann als **starker Hinweis** (keinesfalls als Beweis!) darauf verstanden werden, dass jedes dieser **formalen Berechnungsmodelle** den Begriff **intuitive Berechenbarkeit** (im Sinn von umfassend und vollständig)

► wahrscheinlich **‘gut’** charakterisiert!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Tatsächlich

...schließt dieser starke Hinweis nicht aus, dass vielleicht noch heute ein mächtigeres formales Berechnungsmodell gefunden wird, das den Begriff der intuitiven Berechenbarkeit umfassender und damit vollständiger und besser charakterisierte.

## Präzedenzfall: Das Berechenbarkeitsmodell

### ► primitiv rekursiver Funktionen

galt bis 1928 als adäquate Charakterisierung des Begriffs intuitiver Berechenbarkeit.

- Jedoch: Echt schwächer als spätere Berechnungsmodelle.
- Beweis: Durch Angabe der offensichtlich berechenbaren, aber nicht primitiv rekursiv darstellbaren Ackermann-Funktion (Wilhelm Ackermann, 1928).

(Zur Definition des Schemas primitiv rekursiver Funktionen siehe z.B.: Wolfram-Manfred Lippe. Funktionale und Applikative Programmierung. eXamen.press, 2009, Kapitel 2.1.2.)

# Die Ackermann-Funktion

...‘berühmtberüchtigtes’ Beispiel einer offensichtlich

- effektiv **berechenbaren**, insbesondere also **intuitiv berechenbaren**, jedoch nicht **primitiv rekursiv** darstellbaren Funktion.

Die **Ackermann-Funktion** (in einer Variante von **Rózsa Péter**, 1935) in **Haskell**-Notation:

```
ack :: (Integer,Integer) -> Integer
ack (m,n)
  | m == 0                = n+1
  | (m > 0) && (n == 0)    = ack (m-1,1)
  | (m > 0) && (n /= 0)    = ack (m-1,ack(m,n-1))
```

# Intuitive Berechenbarkeit: Allgemein genug?

...orthogonal zur Suche nach einer

- ▶ umfassenden und vollständigen Formalisierung des Begriffs intuitiver Berechenbarkeit

ist die Frage nach der

- ▶ Angemessenheit des Begriffs intuitiver Berechenbarkeit selbst.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Warum?

Die Auffassung **intuitiver Berechenbarkeit** als Existenzfrage

- ▶ “einer **irgendwie machbaren effektiven mechanischen Methode**, die zu jedem **gültigen** Argumentwert in **endlich vielen Schritten** den Funktionswert konstruiert und für alle anderen Argumentwerte entweder mit einem **speziellen Fehlerwert** oder **nie abbricht**.”

induziert eine

- ▶ **funktionsorientierte** Vorstellung von **Algorithmus**

die Berechnungsmodellen wie dem  **$\lambda$ -Kalkül** und anderen zugrundeliegt und weitergehend implizit die Problemtypen festlegt, die überhaupt als

- ▶ **Berechenbarkeitsproblem**

aufgefasst werden (können).

# Interaktionslose Verarbeitung (1)

...aus Maschinensicht entspricht der funktionsorientierten Algorithmusauffassung eine interaktionslose stapelartige Verarbeitungs- und Berechnungssicht:

► Eingabe  $\rightsquigarrow$  endl. Verarbeitung/Berechnung  $\rightsquigarrow$  Ausgabe

die sich in der Arbeits- und Auswertungsweise der

► Turing-Maschine

► früher automatischer Rechenanlagen (vulgo: Computer)

ebenso findet...

# Interaktionslose Verarbeitung (2)

...wie in der Auswertungsweise unserer bisher betrachteten

► Haskell-Programme.

Nach Bereitstellung der Eingabedaten:

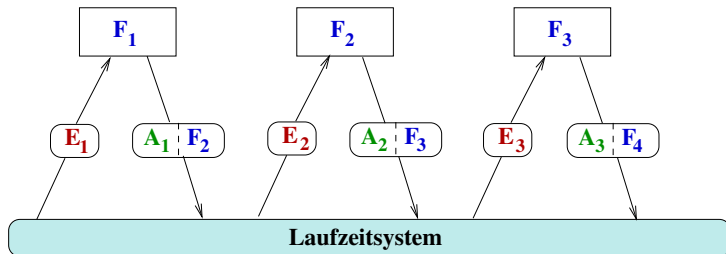


Peter Pepper. *Funktionale Programmierung*.  
Springer-Verlag, 2003, S. 245.

...findet **Interaktion** zwischen Anwender und Programm nicht mehr statt.

# Interaktionsbehaftete Verarbeitung

...**Interaktion** zwischen Anwender und Programm über die Bereitstellung von Eingabedaten hinaus ist für heutige konkrete Rechner jedoch kennzeichnend, auch für **Haskell**-Programme (siehe **Kap. 15**, Ein- und Ausgabe):



Peter Pepper. *Funktionale Programmierung*.  
Springer-Verlag, 2003, S. 253.



# Naheliegende Fragen (1)

Sind **Leistungen**, **Aktivitäten**, **Tätigkeiten**, die

- ▶ Betriebssysteme
- ▶ (Eingebettete) Steuerungssysteme
- ▶ Nebenläufige Systeme, Web-Services, das Internet
- ▶ Graphische Benutzerschnittstellen
- ▶ ...

erbringen oder

- ▶ Fahrzeuge autonom ihren Weg im realen Straßenverkehr zu vorgegebenen Zielen finden lassen
- ▶ ...

durch den **funktionsorientierten** Begriff **intuitiver Berechenbarkeit** gedeckt, d.h. vor- und darstellbar mit einmaliger Eingabedatenbereitstellung **ohne weitere Interaktion**?

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Naheliegende Fragen (2)

...oder sind dies Aufgaben qualitativ anderer, **nichtfunktionaler** Art? Im Fall von:

...**Betriebs-** und **Steuerungssystemen**:

- ▶ Ist die Berechnung, Verarbeitung endlich? Terminiert sie?
- ▶ **Welche Funktion wird berechnet?**

...**Webservices**, dem **Internet**:

- ▶ Können Systeme mit flexibel hinzutretenden und ebenso wieder verschwindenden Komponenten als statisch angesehen werden? Wenn ja, in welchem Sinn?
- ▶ **Welche Funktion wird berechnet?**

...**autonomen Fahrzeugen**:

- ▶ Wie sehen Ein- und Ausgabe aus?
- ▶ **Welche Funktion wird berechnet?**

**Interaktion** scheint für Aufgaben dieser Art unverzichtbar.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Naheliegende Fragen (3)

...ändert **Hinzunahme von Interaktion** das Verständnis von **Berechnung** und **Berechenbarkeit** möglicherweise ähnlich grundlegend wie Ackermanns Funktion?

Angestoßen wurden diese Fragen und Untersuchungen hierzu besonders durch:

- ▶ Peter Wegner. **Why Interaction is More Powerful Than Algorithms**. Communications of the ACM 40(5):81-91, 1997.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Was heißt Berechnung, was berechenbar?

Sind Antworten wie z.B. von [Martin Davis](#) in:

- ▶ [Martin Davis. What is a Computation?](#) Kapitel in Lynn A. Steen (Hrsg.), Mathematics Today – Twelve Informal Essays. Springer-V., 241-268, 1978.

...ausreichend? Müssen sie angepasst u. weiterentwickelt werden angesichts [massiv parallelen](#), [verteilten](#), [interaktiven](#), [asynchronen](#), [analogen](#), [hybriden Rechnens](#), [Echtzeitrechnens](#), [Rechnens mit neuronalen Netzen](#), [chemischen Reaktionsnetzwerken](#), [Quanten-](#), [Nano-](#), [DNS-](#), [molekul.](#), [enzym.](#), [bakt. Rechn.](#),...?

- ▶ [Luca Cardelli. Programming with Chemical Reactions.](#) VCLA-Kolloquiumsvortrag an der TU Wien, 22.11.2018.

"Chemical reactions have been widely used to describe natural phenomena, but increasingly we are capable to use them to prescribe physical interaction, e.g. in DNA computing. Thus, chemical reaction networks can be used as programs that can be physically realized to produce and control molecular arrangements. Because of their relative simplicity and familiarity, and more subtly because of their computational power, they are quickly becoming a paradigmatic "programming language" for bioengineering. We discuss what can be programmed with chemical reactions, and how these programs can be physically realized."

# Berechnungsbegriffe und -paradigmen

Ein Klassiker:

- ▶ John McCarthy. [A Basis for a Mathematical Theory of Computation](#). In *Computer Programming and Formal Systems*, Paul Braffort, David Hirschberg (Hrsg.), North-Holland, 33-70, 1963.

Neue Auffassungen:

- ▶ S. Barry Cooper, Benedikt Löwe, Andrea Sorbi (Hrsg.). [New Computational Paradigms: Changing Conceptions of What is Computable](#). Springer-V., 2008.

Als Einführung gut geeignet:

- ▶ Ian Horswill. [What is Computation?](#) Crossroads, the ACM Magazine for Students 18(3):8-14, 2012.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Schwache vs. starke Church/Turing-These (1)

...im Sinn von **Peter Wegner** auf den Punkt gebracht: Gilt die Church/Turing-These

...im **schwachen** Sinn:

- Wann immer eine (mathematische) Funktion intuitiv berechenbar ist, d.h. wann immer es eine effektive mechanische Methode für ihre Berechnung gibt, dann kann sie von einer Turing-Maschine, im  $\lambda$ -Kalkül berechnet werden.

...oder im **starken** Sinn:

- Was immer eine 'Berechnungsmaschine' ('Computer') (mithilfe auch von Interaktion) berechnen kann, kann von einer Turing-Maschine, im  $\lambda$ -Kalkül berechnet werden, d.h. wann immer (über (mathematische) Funktionen hinaus) eine Aufgabe als Berechnung ausgedrückt und verstanden werden kann, kann sie von einer Turing-Maschine, im  $\lambda$ -Kalkül berechnet werden.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Schwache vs. starke Church/Turing-These (2)

...eine 'für' und 'wider' untersuchte Frage:

- ▶ Michael Prasse, Peter Rittgen. [Why Church's Thesis Still Holds. Some Notes on Peter Wegner's Tracts on Interaction and Computability](#). The Computer Journal 41(6):357-362, 1998.
- ▶ Peter Wegner, Eugene Eberbach. [New Models of Computation](#). The Computer Journal 47(1):4-9, 2004.
- ▶ Paul Cockshott, Greg Michaelson. [Are There New Models of Computation? Reply to Wegner and Eberbach](#). The Computer Journal 50(2):232-247, 2007.
- ▶ Dina Q. Goldin, Peter Wegner. [The Interactive Nature of Computing: Refuting the Strong Church-Rosser Thesis](#). Minds and Machines 18(1):17-38, 2008.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Schwache vs. starke Church/Turing-These (3)

- ▶ Peter Wegner, Dina Q. Goldin. *The Church-Turing Thesis: Breaking the Myth*. In Proceedings of the 1st Conference on Computability in Europe – New Computational Paradigms (CiE 2005), Springer-V., LNCS 3526, 152-168, 2005.
- ▶ Martin Davis. *The Church-Turing Thesis: Consensus and Opposition*. In Proceedings of the 2nd Conference on Computability in Europe – Logical Approaches to Computational Barriers (CiE 2006), Springer-V., LNCS 3988, 125-132, 2006.
- ▶ ...

Wer richtig erkennen will, muss zuvor  
in richtiger Weise gezweifelt haben.

Aristoteles (384 - 322 v.Chr.)  
griech. Philosoph

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12



# Interaktion und Berechenbarkeit

...hat **Interaktion** das Potential zu einer neuen **Ackermannfunktion**-vergleichbaren **Weltsichtänderung**? Eine offene Frage...

Ich weiß, das klingt alles sehr kompliziert.

Fred Sinowatz (1929-2008)

österr. Politiker und Staatsmann, Bundeskanzler

Es ist nicht leicht zu begreifen,  
dass man nicht begreift.

Marie von Ebner-Eschenbach (1830-1916)

österr. Schriftstellerin

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Untersuchung und Diskurs

...gehen weiter; eigenes Verständnis von Voraussetzungen und Implikationen der 'für'- und 'wider'-Argumente sind gefordert.

Einen **kompakten Einstieg** in das Themenfeld zusammen mit Verweisen auf relevante Arbeiten bieten:

- ▶ B. Jack Copeland, Eli Dresner, Diane Proudfoot, Oron Shagrir. **Viewpoint: Time to Reinspect the Foundations? Questioning if Computer Science is Outgrowing its Traditional Foundations.**

Communications of the ACM 59(11):34-36, 2016.

- ▶ B. Jack Copeland, Oron Shagrir. **The Church-Turing Thesis: Logical Limit or Breachable Barrier?**

Communications of the ACM 62(1):66-74, 2019.

...siehe **Anhang B** für weitere Literaturhinweise.

**Nur Beharrung führt zum Ziel.**

Friedrich von Schiller (1759-1805)

dt. Schriftsteller

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Man kann viel, wenn man sich nur recht viel zutraut.

Wilhelm von Humboldt (1767-1835)  
dt. Philosoph und Politiker

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Zurück zum $\lambda$ -Kalkül (1)

## Der $\lambda$ -Kalkül

- ▶ geht zurück auf **Alonzo Church** (1936).
- ▶ ist erdacht als **formales Berechnungsmodell** (neben anderen), um Fragen der Natur von Berechnung und dessen, was berechnet werden kann, zu fassen u. zu untersuchen.
- ▶ formalisiert einen **Berechnungsbegriff** über Paaren, Listen, Bäumen, auch potentiell unendlichen, über Funktionen höherer Ordnung, etc.
- ▶ zeichnet sich in diesem Sinne durch **größere Praxisnähe** als (einige) andere formale Berechnungsmodelle aus.
- ▶ ist mit der Erfindung von **Rechenanlagen** und **Programmiersprachen** geworden
  - ▶ zur Grundlage aller **funktionalen Programmiersprachen**.
  - ▶ zum Bindeglied **funktionaler Hochsprachen** und **maschinennaher Implementierungen**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Zurück zum $\lambda$ -Kalkül (2)

Der  $\lambda$ -Kalkül zeichnet sich dabei aus durch:

- ▶ **Einfachheit**

...wenige syntaktische Konstrukte, einfache Semantik.

- ▶ **Ausdruckskraft**

...Turing-mächtig, alle 'intuitiv berechenbaren' Funktionen im  $\lambda$ -Kalkül ausdrückbar.

# Zurück zum $\lambda$ -Kalkül (3)

...wichtige Anwendungsfelder des  $\lambda$ -Kalküls.

Ursprünglich:

- ▶ **Berechenbarkeitstheorie:** Berechenbarkeitsbegriff, Grenzen der Berechenbarkeit.

Später hinzugekommen:

- ▶ **Entwurf von Programmiersprachen und Programmiersprachkonzepten:** Funktionale Programmiersprachen, Typsysteme, Polymorphie,...
- ▶ **Semantik von Programmiersprachen:** Denotationelle Semantik, Bereichstheorie (engl. domain theory),...

# Reiner $\lambda$ -Kalkül und angewandte $\lambda$ -Kalküle

## Reiner $\lambda$ -Kalkül

- ▶ Reduziert auf das ‘absolut Notwendige’, bedeutsam und angemessen besonders für grundlegende Untersuchungen zu Fragen der Berechenbarkeit, [Berechenbarkeitstheorie](#).

## Angewandte $\lambda$ -Kalküle

- ▶ Syntaktisch angereicherte Varianten des reinen  $\lambda$ -Kalküls, [praxis-](#) und [programmiersprachennäher](#).

## Extrem angereicherte angewandte $\lambda$ -Kalküle

- ▶ **Funktionale Programmiersprachen.**

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Kapitel 12.2

## Syntax des reinen $\lambda$ -Kalküls

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1000/17






# Syntax von $\lambda$ -Ausdrücken (2)

- **Applikation:** Sind  $f$  und  $e$  aus  $E$ , so ist auch  $(f\ e)$  in  $E$ .  
**Sprechweise:** Applikation oder Anwendung von  $f$  auf  $e$ ;  $f$  heißt auch **Rator**,  $e$  auch **Rand**.

Bsp.:  $((\lambda x. (x\ x))\ y), \dots$



# Syntax in BNF-Notation

...Ausdruckssyntax kompakt in Backus-Naur-Form (BNF):

$e$	$::=$	$x$	(Namen)
	$::=$	$\lambda x.e$	((Funktions-) Abstraktion)
	$::=$	$e e$	((Funktions-) Applikation)
	$::=$	$(e)$	(Klammerung)

**Informell:** Der  $\lambda$ -Kalkül bietet Namen, (anonyme) Funktionen und Funktionsanwendungen, durch Klammerung auch in geschachtelter Form.

**Anmerkung:** Statt Name aus N sagt man oft auch Variable aus V und identifiziert N und V miteinander.

# Vereinbarungen, Konventionen

Überflüssige Klammern können weggelassen werden. Es gilt:

- Rechtsassoziativität für  $\lambda$ -Sequenzen in Abstraktionen.

Beispiele:

- $\lambda x. \lambda y. \lambda z. (x (y z))$  steht kurz für  $(\lambda x. (\lambda y. (\lambda z. (x (y z)))))$
- $\lambda x. e$  steht kurz für  $(\lambda x. e)$

- Linksassoziativität für Applikationssequenzen.

Beispiele:

- $e_1 e_2 e_3 \dots e_n$  steht kurz für  $(\dots ((e_1 e_2) e_3) \dots e_n)$
- $e_1 e_2$  steht kurz für  $(e_1 e_2)$

Der Rumpf einer  $\lambda$ -Abstraktion ist der längstmögliche dem Punkt folgende  $\lambda$ -Ausdruck.

Beispiel:  $\lambda x. e f$  steht kurz für  $\lambda x. (e f)$ , nicht  $(\lambda x. e) f$

# Bindungsbereich, Gültigkeitsbereich v. Namen

...in  $\lambda$ -Ausdrücken.

- Der **Bindungsbereich** der gebundenen Variablen einer  $\lambda$ -Abstraktion ist der **Rumpf** der  $\lambda$ -Abstraktion.

$$\underbrace{(\lambda f. \lambda g. \lambda h. f (g h))}_{\text{Bindungsbereich von } f} \underbrace{\lambda g. ((\lambda g. g) g))}_{\text{Bindungsbereich von } g}$$

Bindungsbereich von  $g$

- Der **Gültigkeitsbereich** der gebundenen Variablen einer  $\lambda$ -Abstraktion ist ihr **Bindungsbereich** abzüglich aller inneren **Bindungsbereiche** mit gleichnamiger gebundener Variable.

$$\underbrace{(\lambda f. \lambda g. \lambda h. f (g h))}_{\text{Gültigkeitsbereich von } f} \underbrace{\lambda g. ((\lambda g. g) g))}_{\text{Gültigkeitsbereich von } g}$$

Gültigkeitsbereich von  $g$

# Freie Variablen, gebundene Variablen (1)

...in  $\lambda$ -Ausdrücken. Sei  $a$  aus  $E$ :

Freie Variablen von  $a$ :

$$\text{frei}(x) = \{x\} \quad \text{wenn } a \equiv x \text{ aus } N$$

$$\text{frei}(\lambda x.e) = \text{frei}(e) \setminus \{x\} \quad \text{wenn } a \equiv \lambda x.e$$

$$\text{frei}(f e) = \text{frei}(f) \cup \text{frei}(e) \quad \text{wenn } a \equiv f e$$

Gebundene Variablen von  $a$ :

$$\text{gebunden}(x) = \emptyset \quad \text{wenn } a \equiv x \text{ aus } N$$

$$\text{gebunden}(\lambda x.e) = \text{gebunden}(e) \cup \{x\} \quad \text{wenn } a \equiv \lambda x.e$$

$$\begin{aligned} \text{gebunden}(f e) &= \text{gebunden}(f) \\ &\quad \cup \text{gebunden}(e) \quad \text{wenn } a \equiv f e \end{aligned}$$

Anm.:  $\equiv$  steht für lexikalisch gleich.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1006/17

# Freie Variablen, gebundene Variablen (2)

Beispiel: Betrachte den  $\lambda$ -Ausdruck  $((\lambda x. (x y)) x)$ .

Gesamtausdruck:

- ▶  $x$  kommt in  $((\lambda x. (x y)) x)$  **frei** und **gebunden** vor.
- ▶  $y$  kommt in  $((\lambda x. (x y)) x)$  **frei** vor, aber nicht gebunden.

Teilausdrücke:

- ▶  $x$  kommt in  $(\lambda x. (x y))$  gebunden vor, aber nicht frei.
- ▶  $x$  kommt in  $(x y)$  und  $x$  **frei** vor, aber nicht gebunden.
- ▶  $y$  kommt in  $(\lambda x. (x y))$ ,  $(x y)$  und  $y$  **frei** vor, aber nicht gebunden.

Beachte: 'Gebunden' und 'frei' sind **nicht** Negationen voneinander (anderenfalls gälte z.B. ' $x$  kommt gebunden in  $y$  vor' oder ' $y$  kommt frei in  $\lambda x. x$  vor').

# Freie, gebundene Variablenvorkommen

...in  $\lambda$ -Ausdrücken:

- ▶ **Definierende** Vorkommen: Jedes Variablenvorkommen unmittelbar nach einem  $\lambda$ .
- ▶ **Angewandte** Vorkommen: Jedes nicht definierende Variablenvorkommen.
- ▶ **Gebunden an**: Relation zwischen Variablenvorkommen und definierenden Variablenvorkommen. Jedes Variablenvorkommen (gleich ob angewandt oder definierend) ist an höchstens ein definierendes Variablenvorkommen gebunden; definierende Vorkommen sind an ihr Vorkommen selbst gebunden.
- ▶ **Freies Variablenvorkommen**: Angewandtes Vorkommen, das an kein definierendes Vorkommen gebunden ist.
- ▶ **Gebundenes Variablenvorkommen**: Vorkommen (gleich ob angewandt oder definierend), das an ein definierendes Vorkommen gebunden ist.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1008/17



# Kapitel 12.3

## Semantik des reinen $\lambda$ -Kalküls

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1009/17

# Vorbereitung

...grundlegend für die **Definition** der **Semantik** von  $\lambda$ -**Ausdrücken** sind:

- ▶ Syntaktische Substitution
- ▶ Konversionsregeln/Reduktionsregeln
- ▶ Reduktionsfolgen/Reduktionsstrategien
- ▶ Normalformen (Existenz/Eindeutigkeit)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

10/17

# Kapitel 12.3.1

## Syntaktische Substitution

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1011/17

# Syntaktische Substitution: Informell

...eine dreistellige **Abbildung**

$$\cdot[\cdot/\cdot] : E \rightarrow E \rightarrow V \rightarrow E$$

zur **bindungsfehlerfreien** Ersetzung frei vorkommender Variablen  $x$  durch einen Ausdruck  $e$  in einem Ausdruck  $e'$ .

**Informell:** Angewendet auf zwei Ausdrücke  $e'$  und  $e$  und eine Variable  $x$  bezeichnet

$$e' [e/x]$$

denjenigen Ausdruck, der aus  $e'$  entsteht, indem **jedes freie** Vorkommen von  $x$  in  $e'$  durch  $e$  **substituiert**, ersetzt wird.

**Beachte:** Die vereinfachende informelle Beschreibung nimmt keinen Bedacht auf mögliche **Bindungsfehler**. Freiheit von Bindungsfehlern stellt die formale Definition **syntaktischer Substitution** sicher.

# Syntaktische Substitution

## Definition 12.3.1.1 (Syntaktische Substitution)

Die **syntaktische Substitution** ist die 3-stellige Abbildung

$\cdot[\cdot/\cdot] : E \rightarrow E \rightarrow V \rightarrow E$  definiert bei Anwendung auf

...**Namensterme** durch:

$$y[e/x] =_{df} \begin{cases} y & \text{falls } y \text{ aus } N \text{ mit } y \neq x \\ e & \text{falls } y \text{ aus } N \text{ mit } y = x \end{cases}$$

...**applikative Terme** durch:

$$(f\ g)[e/x] =_{df} (f[e/x])\ (g[e/x])$$

...**Abstraktionsterme** durch:

$$(\lambda y.f)[e/x] =_{df} \begin{cases} \lambda y.f & \text{falls } y = x \\ \lambda y.(f[e/x]) & \text{falls } y \neq x \wedge y \notin \text{frei}(e) \\ \lambda z.((f[z/y])[e/x]) & \text{falls } y \neq x \wedge y \in \text{frei}(e), \\ & \text{wobei } z \text{ frisch aus } N: z \notin \text{frei}(e) \cup \text{frei}(f) \\ & \text{(Vermeidung von Bindungsfehlern!)} \end{cases}$$

# Beispiele (1)

...zur Anwendung syntaktischer Substitution:

$$\blacktriangleright ((x \textcolor{blue}{y}) (\textcolor{blue}{y} z)) [(a \textcolor{blue}{b})/\textcolor{blue}{y}] = ((x (a \textcolor{blue}{b})) ((a \textcolor{blue}{b}) z))$$

$$\blacktriangleright \lambda x. (x \textcolor{blue}{y}) [(a \textcolor{blue}{b})/\textcolor{blue}{y}] = \lambda x. (x (a \textcolor{blue}{b}))$$

$$\blacktriangleright \lambda \textcolor{red}{x}. (\textcolor{red}{x} y) [(a \textcolor{blue}{b})/\textcolor{blue}{x}] = \lambda \textcolor{red}{x}. (\textcolor{red}{x} y)$$

## Beispiele (2)

Ein **Bindungsfehler** entsteht bei naiver Anwendung **syntaktischer Substitution**, wenn ein Ausdruck mit einer freien Variable in den Gültigkeitsbereich einer gebundenen Variable gleichen Namens eingesetzt wird:

- $\lambda x. (x\ y) [(x\ b)/y] \xrightarrow{\text{naiv}} \lambda x. (x\ (x\ b))$ : **Bindungsfehler!**  
...naiv ohne **Umbenennung** angewendet ist **x** eingefangen!

Korrekt mit **Umbenennung** angewendet gibt es **keinen Bindungsfehler**:

- $\lambda x. (x\ y) [(x\ b)/y] = \underbrace{\lambda z. ((x\ y)[z/x])}_{\substack{\text{x frei in (x b) Umbenennung von x in z} \\ \text{Umbenannt}}} [(x\ b)/y]$   
 $= \underbrace{\lambda z. (z\ y)}_{\text{Umbenannt}} [(x\ b)/y]$   
 $= \lambda z. (z\ (\underbrace{x}_b\ b))$

**Kein Bindungsfehler**: **x** in **(x b)** bleibt **frei**!

# Kapitel 12.3.2

## Konversionsregeln

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1016/17



# $\lambda$ -Konversionsregeln, $\lambda$ -Konversionen

...die  $\lambda$ -Konversionsregeln führen abgestützt auf syntaktische Substitution zu einer operationellen Semantik für  $\lambda$ -Ausdrücke in Form maximaler Ausdrucksvereinfachung:

## Definition 12.3.2.1 ( $\lambda$ -Konversionsregeln)

- $\alpha$ -Konversion (Umbenennung von Parametern)

$$\lambda x. e \longleftrightarrow \lambda y. e[y/x], \text{ wobei } y \notin \text{frei}(e)$$

- $\beta$ -Konversion (Funktionsanwendung)

$$(\lambda x. f) e \longleftrightarrow f[e/x]$$

- $\eta$ -Konversion (Elimination redundanter Funktion)

$$\lambda x. (e x) \longleftrightarrow e, \text{ wobei } x \notin \text{frei}(e)$$

# Zur Anwendung der Konversionsregeln

## $\alpha$ -Konversion zur

- konsistenten Umbenennung von Parametern von  $\lambda$ -Abstraktionen (zur Vermeidung von Bindungsfehlern (s.u.)!).

## $\beta$ -Konversion zur

- Anwendung einer  $\lambda$ -Abstraktion auf ein Argument.

## $\eta$ -Konversion zur

- Elimination unnötiger  $\lambda$ -Abstraktionen.

Erinnerung: Naiv ohne  $\alpha$ -Konversion angewendet, kann die Substitution der  $\beta$ -Konversion Bindungsfehler verursachen, wie im nachstehenden Beispiel illustriert:

Bsp.:  $(\lambda x. (\lambda y. x \ y)) \ (y \ z) \longrightarrow (\lambda y. x \ y)[(y \ z)/x] \longrightarrow (\lambda y. (y \ z) \ y)$   
(ohne  $\alpha$ -Konversion ist  $y$  eingefangen: Bindungsfehler!)

...korrekt angewendet: Keine Bindungsfehler dank  $\alpha$ -Konversion!

# Konversionsregeln, Reduktionsregeln

...Sprechweisen im Zusammenhang mit Konversionsregeln:

- ▶ Von links nach rechts angewendet: Reduktion.
- ▶ Von rechts nach links angewendet: Abstraktion.

Genauer:

- ▶ Von links nach rechts gerichtete Anwendungen der  $\beta$ - und  $\eta$ -Konversion heißen  $\beta$ -Reduktion und  $\eta$ -Reduktion.
- ▶ Von rechts nach links gerichtete Anwendungen der  $\beta$ -Konversion heißen  $\beta$ -Abstraktion.

Entsprechend spricht man von  $\beta$ - und  $\eta$ -Reduktionsregeln, um auszudrücken, dass die entsprechenden Konversionsregeln nur von links nach rechts angewendet werden.

# Kapitel 12.3.3

## Reduktionsfolgen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1020/17

# Reduktionsfolgen, -strategien, Normalform

Eine **Reduktionsfolge** für einen  $\lambda$ -Ausdruck

- ▶ ist eine endliche oder nicht endliche Folge von  $\beta$ -,  $\eta$ -Reduktionen und  $\alpha$ -Konversionen.
- ▶ heißt **maximal**, wenn höchstens noch  $\alpha$ -Konversionen anwendbar sind.

Ein  $\lambda$ -Ausdruck  $e$  ist in **Normalform**, wenn

- ▶  $e$  durch  $\beta$ -,  $\eta$ -Reduktion nicht weiter reduzierbar ist.

(Praktisch relevante) **Reduktionsstrategien** sind

- ▶ Normale (Reduktions-) Ordnung (linke-äußerst)
- ▶ Applikative (Reduktions-) Ordnung (linke-innerst)

# Beispiele zu Reduktionsfolgen, -strategien (1)

## Beispiel 1: Applikative Ordnung

$$\underbrace{((\lambda z. \lambda y. (z y))}_{\text{Rator}} \underbrace{(\lambda x. x))}_{\text{Rand}} (\lambda s. (s s))$$

$$(\beta\text{-Reduktion}) \longrightarrow \underbrace{(\lambda y. ((\lambda x. x) y))}_{\text{Rator}} \underbrace{(\lambda s. (s s))}_{\text{Rand}}$$

$$(\beta\text{-Reduktion}) \longrightarrow \underbrace{(\lambda x. x)}_{\text{Rator}} \underbrace{(\lambda s. (s s))}_{\text{Rand}}$$

$$(\beta\text{-Reduktion}) \longrightarrow \lambda s. (s s)$$

...fertig, Normalform erreicht: Keine  $\beta$ -,  $\eta$ -Reduktion mehr anwendbar.

# Beispiele zu Reduktionsfolgen, -strategien (2)

## Beispiel 2: Applikative Ordnung

$$((\lambda x. \lambda y. x \ y) ((\underbrace{(\lambda x. \lambda y. x \ y)}_{\text{Rator}} \underbrace{a}_{\text{Rand}}) b)) \ c$$

$$(\beta\text{-Reduktion}) \longrightarrow (\lambda x. \lambda y. x \ y) ((\underbrace{(\lambda y. a \ y)}_{\text{Rator}} \underbrace{b}_{\text{Rand}})) \ c$$

$$(\beta\text{-Reduktion}) \longrightarrow ((\underbrace{(\lambda x. \lambda y. x \ y)}_{\text{Rator}} \underbrace{(a \ b)}_{\text{Rand}})) \ c$$

$$(\beta\text{-Reduktion}) \longrightarrow (\underbrace{(\lambda y. (a \ b) \ y)}_{\text{Rator}} \underbrace{c}_{\text{Rand}})$$

$$(\beta\text{-Reduktion}) \longrightarrow (a \ b) \ c$$

...fertig, Normalform erreicht: Keine  $\beta$ -,  $\eta$ -Reduktion mehr anwendbar.

# Beispiele zu Reduktionsfolgen, -strategien (3)

## Beispiel 2': Normale Ordnung

$$\begin{aligned} & \underbrace{((\lambda x. \lambda y. x \ y))}_{\text{Rator}} \underbrace{(((\lambda x. \lambda y. x \ y) \ a) \ b))}_{\text{Rand}} \ c \\ (\beta\text{-Reduktion}) & \longrightarrow \underbrace{(\lambda y. (((\lambda x. \lambda y. x \ y) \ a) \ b) \ y))}_{\text{Rator}} \underbrace{c}_{\text{Rand}} \\ (\beta\text{-Reduktion}) & \longrightarrow (((\underbrace{(\lambda x. \lambda y. x \ y)}_{\text{Rator}} \underbrace{a}_{\text{Rand}}) \ b) \ c) \\ (\beta\text{-Reduktion}) & \longrightarrow ((\underbrace{(\lambda y. a \ y)}_{\text{Rator}} \underbrace{b}_{\text{Rand}}) \ c) \\ (\beta\text{-Reduktion}) & \longrightarrow (a \ b) \ c \end{aligned}$$

...fertig, Normalform erreicht: Keine  $\beta$ -,  $\eta$ -Reduktion mehr anwendbar.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1024/17



# Kapitel 12.3.4

## Normalformen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1025/17

# Existenz von Normalformen

...Normalformen existieren nicht notwendig; nicht jeder  $\lambda$ -Ausdruck

- ▶ besitzt oder ist in Normalform konvertierbar.

Beispiel: Der Ausdruck

$$\underbrace{\lambda x.(x\ x)}_{\text{Rator}} \underbrace{\lambda x.(x\ x)}_{\text{Rand}} \longrightarrow \lambda x.(x\ x) \lambda x.(x\ x) \longrightarrow \dots$$

...reproduziert sich endlos: Normalform existiert nicht!

# Terminierung von Reduktionsfolgen

...Reduktionsfolgen terminieren nicht notwendig mit einem

- $\lambda$ -Ausdruck in Normalform, selbst wenn eine Normalform existiert.

Beispiel:

$$\underbrace{(\lambda x. y)}_{\text{Rator}} \underbrace{(\lambda x. (x x) \lambda x. (x x))}_{\text{Rand}} \longrightarrow y$$

Normale Reduktionsordnung terminiert in einem Schritt:  
Normalform existiert!

$$\underbrace{(\lambda x. y)}_{\text{Rator}} \underbrace{(\lambda x. (x x) \lambda x. (x x))}_{\text{Rand}} \longrightarrow (\lambda x. y) \underbrace{(\lambda x. (x x))}_{\text{Rator}} \underbrace{\lambda x. (x x)}_{\text{Rand}}$$

$\longrightarrow \dots$

Applikative Reduktionsordnung terminiert nicht, obwohl  
Normalform existiert!

# Church/Rosser-Theoreme

Seien  $e_1, e_2$  zwei  $\lambda$ -Ausdrücke.

## Theorem 12.3.4.1 (Konfluenz-, Diamant-, Rauteneig.)

Wenn  $e_1, e_2$  ineinander konvertierbar sind, d.h.  $e_1 \longleftrightarrow e_2$ , dann gibt es einen gemeinsamen  $\lambda$ -Ausdruck  $e$ , zu dem  $e_1, e_2$  reduziert werden können, d.h.  $e_1 \longrightarrow^* e$  und  $e_2 \longrightarrow^* e$ .

**Informell:** Wenn eine Normalform existiert, dann ist sie (bis auf  $\alpha$ -Konversion) eindeutig bestimmt!

## Theorem 12.3.4.2 (Standardisierung)

Wenn  $e_1$  zu  $e_2$  mit einer endlichen Reduktionsfolge reduzierbar ist, d.h.  $e_1 \longrightarrow^* e_2$ , und  $e_2$  in Normalform ist, dann führt auch die normale Reduktionsfolge von  $e_1$  nach  $e_2$ .

**Informell:** Die normale Reduktionsordnung terminiert am häufigsten, so oft wie überhaupt möglich!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1028/17

# Folgerungen

...aus den Church/Rosser-Theoremen (Alonzo Church, John Barkley Rosser (1936)):

- ▶ Theorem 12.3.4.1 garantiert, dass die Normalform eines  $\lambda$ -Ausdrucks (bis auf  $\alpha$ -Konversionen) eindeutig bestimmt ist, wenn sie existiert;  $\lambda$ -Ausdrücke in Normalform lassen sich (abgesehen von  $\alpha$ -Konversionen) nicht mehr weiter reduzieren, vereinfachen.
- ▶ Theorem 12.3.4.2 garantiert, dass die normale Reduktionsordnung mit der Normalform terminiert, wenn es irgendeine Reduktionsfolge mit dieser Eigenschaft gibt, d.h. die normale Reduktionsordnung terminiert mindestens so häufig wie jede andere Reduktionsstrategie, mit hin am häufigsten.

Omnia viae Romam ducunt.  
Alle Wege führen zum Ergebnis  
(wenn sie denn zum Ergebnis führen).

lat., sprichwörtl., abgewandelt

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1029/17

# Kapitel 12.3.5

## Semantik von $\lambda$ -Ausdrücken

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1030/17

# Semantik von $\lambda$ -Ausdrücken

...die Church/Rosser-Theoreme und ihre Garantien legen nahe, die Semantik (oder Bedeutung) der Ausdrücke des reinen  $\lambda$ -Kalküls in folgender Weise festzulegen:

## Definition 12.3.5.1 (Semantik von $\lambda$ -Ausdrücken)

Sei  $e$  ein  $\lambda$ -Ausdruck. Die Semantik von  $e$  ist

- ▶ seine (bis auf  $\alpha$ -Konversionen) eindeutig bestimmte Normalform, wenn sie existiert; die Normalform ist zugleich der Wert von  $e$ .
- ▶ undefiniert, wenn die Normalform nicht existiert.

# Determiniertheit, Turingmächtigkeit

## Lemma 12.3.5.2 (Determiniertheit)

Wenn ein  $\lambda$ -Ausdruck in einen  $\lambda$ -Ausdruck in Normalform konvertierbar ist, dann führt jede terminierende Reduktionsfolge des  $\lambda$ -Ausdrucks (bis auf  $\alpha$ -Konversion) zu dieser Normalform, d.h. das Resultat jeder terminierenden Reduktionsfolge ist (bis auf  $\alpha$ -Konversion) determiniert.

## Theorem 12.3.5.3 (Turingmächtigkeit)

Eine Funktion ist im  $\lambda$ -Kalkül genau dann berechenbar, wenn sie Turing-berechenbar, Markov-berechenbar, etc., ist, d.h. im  $\lambda$ -Kalkül sind alle Funktionen berechenbar, die Turing-berechenbar, Markov-berechenbar, etc., sind und umgekehrt.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1032/17



# Kapitel 12.3.6

## Rekursion vs. Y-Kombinator

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1033/17

# $\lambda$ -Kalkül und Rekursion

...nicht füreinander gemacht. Betrachte die rekursive Haskell-Rechenvorschrift `fac`:

Argumentbehaftet:

```
fac n = if n == 0 then 1 else n * fac (n - 1)
```

Argumentfrei:

```
fac = \n -> if n == 0 then 1 else n * fac (n - 1)
```

Im  $\lambda$ -Kalkül stellt sich folgendes Problem:

- ▶  $\lambda$ -Abstraktionen sind `anonym` und können deshalb `nicht (rekursiv) aufgerufen` werden (auch nicht in Haskell):  
 $\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \dots ??? \dots$
- ▶ `Rekursive Aufrufe` wie im Rumpf von `fac` lassen sich deshalb `nicht` ohne weiteres im (reinen)  $\lambda$ -Kalkül ausdrücken.

# Kunstgriff: Kombinatoren, Y-Kombinator

... $\lambda$ -Terme ohne freie Variablen heißen Kombinatoren.

Ein spezieller Kombinator: Der Y-Kombinator:

$$Y = \lambda f. (\lambda x. (f (x x)) \lambda x. (f (x x)))$$

...ein Kombinator mit Selbstanwendung:

$$Y = \lambda f. (\underbrace{\lambda x. (f (x x))}_{\text{Rator}} \underbrace{\lambda x. (f (x x))}_{\text{Rand}})$$

...Rator ident mit Rand: Selbstanwendung!

# Schlüsseleigenschaft des Y-Kombinators

...Selbstreproduktion!

Für  $e$   $\lambda$ -Ausdruck ist  $(Y\ e)$  zu  $(e\ (Y\ e))$  konvertierbar:

$$\begin{aligned} Y\ e &\longleftrightarrow \underbrace{(\lambda f. (\lambda x. (f\ (x\ x)))\ \lambda x. (f\ (x\ x)))}_{= Y} e \\ &\longrightarrow \underbrace{\lambda x. (e\ (x\ x))\ \lambda x. (e\ (x\ x))}_{= Y\ e} \\ &\longrightarrow e\ (\underbrace{\lambda x. (e\ (x\ x))\ \lambda x. (e\ (x\ x))}_{= Y\ e}) \\ &\longleftrightarrow \underbrace{e}_{\text{Kopie}}\ (\underbrace{Y\ e}_{\text{Selbstreproduktion}}) \end{aligned}$$

...Selbstreproduktion plus Kopie des Arguments  $e$ !

# Der Y-Kombinator

...erlaubt **Rekursion** auf

► **Kopieren** zurückzuführen und zu realisieren.

Idee: Überführe eine **rekursive** Darstellung von **f** in eine **nicht-rekursive** Darstellung, die den **Y-Kombinator** verwendet:

$$\begin{aligned} f &= \dots f \dots && \text{(Rekursive Darstellung von } f) \\ \rightsquigarrow f &= \lambda f. (\dots f \dots) f && (\lambda\text{-Abstraktion}) \\ \rightsquigarrow f &= \underbrace{Y \lambda f. (\dots f \dots)}_{\text{'Y e'}} && \text{(Nichtrekursive Darstellung von } f) \end{aligned}$$

**Übungsaufgabe:** Vergleiche den Effekt des **Y-Kombinators** mit der **Kopierregelsemantik** prozeduraler Programmiersprachen.

# Übungsaufgabe 12.3.6.1

## Anwendung des Y-Kombinators

Betrachte die rekursionsfreie Darstellung von **fac** mit **Y-Kombinator**:

$$\text{fac} = \mathbf{Y} \ \lambda f. (\lambda n. \text{if } n == 0 \text{ then } 1 \text{ else } n * f(n - 1))$$

Zeige durch **nachrechnen**, dass sich der Term **(fac 1)** auf den in Normalform vorliegenden Term **1** reduzieren lässt:

$$\blacktriangleright \text{fac } 1 \longrightarrow \dots \longrightarrow 1$$

**Überprüfe** dabei, dass durch den **Y-Kombinator**

- ▶ Rekursion auf wiederholtes Kopieren zurückgeführt wird.

# Kapitel 12.4

## Angewandte $\lambda$ -Kalküle

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1039/17

# Angewandte $\lambda$ -Kalküle

...sind syntaktisch angereicherte Varianten des reinen  $\lambda$ -Kalküls.

In Ausdrücken angewandter  $\lambda$ -Kalküle können

- Konstanten, Funktionsnamen, 'übliche' Operatoren ähnlich wie Namen auftreten und an die Seite von  $\lambda$ -Abstraktionen treten:

42, 3.14, true, false, +, \*, -, fac, binom, ...

- neue Ausdrücke als Abkürzungen eingeführt und verwendet werden:

cond e e<sub>1</sub> e<sub>2</sub> , if e then e<sub>1</sub> else e<sub>2</sub>, ...

- Typen auftreten, Ausdrücke getypt sein:

42 : IN, 3.14 : IR, true : IBool, ...

- ...



# Wohlgeformte Ausdrücke

...angewandter  $\lambda$ -Kalküle können dann auch

- Applikative Terme wie

$2+3$ ,  $\text{fac } 3$ ,  $\text{fib } (2+3)$ ,  $\text{binom } x \ y$ ,  $((\text{binom } x) \ y)$ , ...

- Abstraktionsterme wie

$\lambda x. (x + x)$ ,  $\lambda x. \lambda y. \lambda z. (x * (y - z))$ ,  
 $(\lambda x. \text{if odd } x \text{ then } x * 2 \text{ else } x \text{ div } 2)$ , ...

sein, für deren Auswertung zusätzliche **Reduktionsregeln** eingeführt werden, sog.:

- $\delta$ -Reduktionen

...zur **Auswertung**, **Reduktion** arithmetischer Ausdrücke, bedingter Ausdrücke, Listenoperationen, etc.

# Beispiel

... $\delta$ -Reduktionsfolge: Unecht 'applikative' Ordnung  
(unecht, da ohne Verzahnung von  $\beta$ -,  $\eta$ - und  $\delta$ -Reduktionen)

$(\lambda x. \lambda y. x * y) ((\lambda x. \lambda y. x + y) 9 5) 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda x. \lambda y. x * y) ((\lambda y. 9 + y) 5) 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda x. \lambda y. x * y) (9 + 5) 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda y. (9 + 5) * y) 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (9 + 5) * 3$

...keine  $\beta$ -,  $\eta$ -Reduktion mehr anwendbar; weiter mit  $\delta$ -Reduktionen:

...  $(9 + 5) * 3$

$(\delta\text{-Reduktion, li}) \longrightarrow 14 * 3$

$(\delta\text{-Reduktion, li}) \longrightarrow 42$

Anm.: Ratoren in rot, Randen in gold; li für linkest-innerst.

# Beispiel'

... $\delta$ -Reduktionsfolge: Applikative Ordnung

$$\begin{aligned} & (\lambda x. \lambda y. x * y) ((\lambda x. \lambda y. x + y) \textcolor{red}{9} \textcolor{gold}{5}) \textcolor{gold}{3} \\ & (\beta\text{-Reduktion, li}) \longrightarrow (\lambda x. \lambda y. x * y) ((\lambda y. \textcolor{red}{9} + y) \textcolor{gold}{5}) \textcolor{gold}{3} \\ & (\beta\text{-Reduktion, li}) \longrightarrow (\lambda x. \lambda y. x * y) (\textcolor{red}{9} + \textcolor{gold}{5}) \textcolor{gold}{3} \\ & (\delta\text{-Reduktion, li}) \longrightarrow (\lambda x. \lambda y. x * y) \textcolor{red}{14} \textcolor{gold}{3} \\ & (\beta\text{-Reduktion, li}) \longrightarrow (\lambda y. \textcolor{red}{14} * y) \textcolor{gold}{3} \\ & (\beta\text{-Reduktion, li}) \longrightarrow \textcolor{red}{14} * \textcolor{gold}{3} \\ & (\delta\text{-Reduktion, li}) \longrightarrow 42 \end{aligned}$$

Anm.: **R**atoren in **rot**, **R**anden in **gold**; **li** für **l**inke**s**t-**i**nn**e**r**s**t.

# Übungsaufgabe 12.4.1

## $\delta$ -Reduktionsfolgen

Gegeben sei der  $\lambda$ -Ausdruck  $e$ :

$$(\lambda x. \lambda. x * y) ((\lambda x. \lambda y. x + y) 9 \ 5) \ 3$$

Ergänze die **applikative  $\delta$ -Reduktionsfolge** für diesen Ausdruck aus dem vorhergehenden Beispiel um

- ▶ die **normale  $\delta$ -Reduktionsfolge**.
- ▶ zwei **weitere** zur **Normalform** führende  **$\delta$ -Reduktionsfolgen** verschieden von der applikativen und normalen Folge.
- ▶ Gibt es darüberhinaus weitere verschiedene zur Normalform führende Reduktionsfolgen für  $e$ ?

# Typisierte $\lambda$ -Kalküle

...ordnen jedem wohlgeformten Ausdruck einen Typ zu, zum Beispiel:

$$3 : \text{Int}$$
$$(*) : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$
$$(\lambda x. 2 * x) : \text{Int} \rightarrow \text{Int}$$
$$(\lambda x. 2 * x) 3 : \text{Int}$$

Dabei treten zwei Schwierigkeiten auf:

1) Ausdrücke mit Selbstanwendung (wie z.B. der Y-Kombinator) können

- ▶ nicht endlich getypt werden, d.h. ihr Typ kann nicht durch einen gewöhnlichen endlichen Typausdruck beschrieben werden.

2) Ausdrücke wie der Y-Kombinator können unmittelbar

- ▶ nicht zur Modellierung von Rekursion verwandt werden.

# Typisierung, Selbstanwendung, Rekursion

...Selbstanwendung im Y-Kombinator:

$$\blacktriangleright Y = \lambda f. (\lambda x. (f (x x)) \lambda x. (f (x x)))$$

verhindert

- ▶ endliche Typisierung von Y in gewöhnlicher Weise.
- ▶ die Modellierung von Rekursion durch kopieren mit-hilfe von Y.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1046/17

# Überwindung

...der **Typisierungsschwierigkeit**:

- ▶ **Rigoros**: Übergang zu **mächtigeren Typsprachen** (**Be-reichstheorie**, **reflexive Bereiche** (engl. **domain theory**, **reflexive domains**)).

...der **Rekursionschwierigkeit**:

- ▶ **Pragmatisch**: Explizite Hinzunahme der **Reduktionsregel**  
 $Y\ e \longrightarrow e\ (Y\ e)$   
zum Kalkül.

# Rechtfertigung, Fundierung

...des Umgangs mit **angereicherten angewandten  $\lambda$ -Kalkülen** und des **pragmatischen** Vorgehens der Reduktionsregelhinzunahme für Rekursion:

Resultate aus der **theoretischen Informatik**, insbesondere

- ▶ Alonzo Church. **The Calculi of Lambda-Conversion**. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941.
- ...u.a. zur Modellierung **ganzer Zahlen**, **Wahrheitswerten**, etc. durch Ausdrücke des **reinen  $\lambda$ -Kalküls**.



- ▶ Der Übergang zu **angewandten  $\lambda$ -Kalkülen** ist aus praktischer Hinsicht sinnvoll und einsichtig; für theoretische Untersuchungen zur Berechenbarkeit (**Berechenbarkeitstheorie**) sind sie kaum relevant.
- ▶ Die Regelhinzunahme zur Rekursionsmodellierung ist aus Effizienzgründen auch **pragmatisch zweckmäßig**.

# Kapitel 12.5

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1050/17

# Zusammenfassung

- ▶ Haskell beruht auf den Grundlagen **typisierter  $\lambda$ -Kalküle**.
- ▶ Übersetzer, Interpretierer prüfen, ob die **Typisierung** von Haskell-Programmen **konsistent**, **wohlgetypt** ist.
- ▶ Programmierer können Typdeklarationen angeben (**aus-sagekräftigere Fehlermeldungen**, **Sicherheit**), müssen aber nicht (bequem, doch u.U. mit unerwarteten Folgen, etwa bei “zufällig” korrekter, aber “ungemeinter” Typisierung: “gemeinte” Typisierung wäre bei Angabe bei der Typprüfung als **inkonsistent** aufgefallen).
- ▶ **Typinformation** (gleich ob angegeben oder nicht) wird vom Übersetzer, Interpretierer **berechnet**, **inferiert**.
- ▶ **Rekursion** kann unmittelbar ausgedrückt werden (**Y-Kombinator** nicht erforderlich).

# Abschließend

...Anekdote zur Entwicklung der  $\lambda$ -Notation anhand der durch eine abstrakte  $\lambda$ -Abstraktion definierten Funktion:

```
fac :: Integer -> Integer
fac = \n -> (if n == 0 then 1 else (n * fac (n - 1)))
```

In Haskell abweichend vom  $\lambda$ -Kalkül also Verwendung von

► “\” und “->” anstelle von “ $\lambda$ ” und “.”

...der Weg dorthin war kurvenreich:

	$\widehat{(n. n + 1)}$	(Churchs handschriftliche Schreibweise)
↪	$(\wedge n. n + 1)$	(Churchs notat. Zugeständnis an Schriftsetzer)
↪	$(\lambda n. n + 1)$	(Pragmatische Umsetzung durch Schriftsetzer)
↪	$(\backslash n \rightarrow n+1)$	(ASCII-Umsetzung in Haskell)

(siehe: Peter Pepper. Funktionale Programmierung in Opal, ML, Haskell und Gofer. Springer-V., 2. Auflage, 2003, S. 22.)

# Kapitel 12.6

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1053/17

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 12 (1)

-  Wilhelm Ackermann. *Zum Hilbertschen Aufbau der reellen Zahlen*. Mathematische Annalen 99:118-133, 1928.
-  Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, Philip Wadler. *The Call-by-Need Lambda Calculus*. In Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), 233-246, 1995.
-  Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised Edn., North-Holland, 1984. (Kapitel 1, Introduction; Kapitel 2, Conversion; Kapitel 3, Reduction; Kapitel 6, Classical Lambda Calculus; Kapitel 11, Fundamental Theorems)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11





Teil V

Kap. 12

12.1

1054/17

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 12 (2)

-  Hendrik P. Barendregt, Erik Barendsen. *Introduction to the Lambda Calculus*. Revised Edn., Technical Report, University of Nijmegen, March 2000.  
<ftp://ftp.cs.kun.nl/pub/CompMath.Found/lambda.pdf>
-  Henrik P. Barendregt, Wil Dekkers, Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2012.
-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 19, Berechenbarkeit und Lambda-Kalkül)
-  Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11






Teil V

Kap. 12

12.1

1055/17

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 12 (3)

-  Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 5, Lambda Calculus)
-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 4, Der Lambda-Kalkül)
-  Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988. (Kapitel 6, Mathematical foundations: the lambda calculus)
-  Robert M. French. *Moving Beyond the Turing Test*. Communications of the ACM 55(12):74-77, 2012.
-  Robin Gandy. *The Confluence of Ideas in 1936*. In Rolf Herken (Hrsg.), *The Universal Turing Machine: A Half-Century Survey*. Springer-V., 2. Auflage, 51-102, 1995.



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 12 (4)



Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004. (Kapitel 1, Introduction; Kapitel 2, Notation and the Basic Theory; Kapitel 3, Reduction; Kapitel 10, Further Reading)







Ian Horswill. *What is Computation?* Crossroads, the ACM Magazine for Students 18(3):8-14, 2012.



Achim Jung. *Berechnungsmodelle*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 73-88, 2006. (Kapitel 2.1, Speicherorientierte Modelle: Turing-Maschinen, Registermaschinen; Kapitel 2.2, Funktionale Modelle: Algebraische Kombinationen, Primitive Rekursion,  $\mu$ -Rekursion,  $\lambda$ -Kalkül)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 12 (5)

-  Stephen C. Kleene. *General Recursive Functions of Natural Numbers*. Mathematische Annalen 112:727-742, 1936.
-  Stephen C. Kleene.  *$\lambda$ -Definability and Recursiveness*. Duke Mathematical Journal 2:340-352, 1936.
-  Stephen C. Kleene. *Origins of Recursive Function Theory*. Annals of the History of Computing 3:52-67, 1981.
-  Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. eXamen.press, 2009. (Kapitel 2.1, Berechenbare Funktionen; Kapitel 2.2, Der  $\lambda$ -Kalkül)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

12.1

1058/17

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 12 (6)



John Maraist, Martin Odersky, David N. Turner, Philip Wadler. *Call-by-name, Call-by-value, call-by-need, and the Linear Lambda Calculus*. Electronic Notes in Theoretical Computer Science 1:370-392, 1995.



John Maraist, Martin Odersky, Philip Wadler. *The Call-by-Need Lambda Calculus*. Journal of Functional Programming 8(3):275-317, 1998.



John Maraist, Martin Odersky, David N. Turner, Philip Wadler. *Call-by-name, Call-by-value, call-by-need, and the Linear Lambda Calculus*. Theoretical Computer Science 228(1-2):175-210, 1999.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11




Teil V

Kap. 12

12.1

1059/17

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 12 (7)

-  John McCarthy. *A Basis for a Mathematical Theory of Computation*. In *Computer Programming and Formal Systems*, P. Braffort, D. Hirschberg (Hrsg.), North-Holland, 33-70, 1963.
-  Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011. (Kapitel 2, Lambda calculus; Kapitel 4.1, Repetition, iteration and recursion; Kapitel 4.3, Passing a function to itself; Kapitel 4.6, Recursion notation; Kapitel 8, Evaluation)
-  William Newman. *Alan Turing Remembered – A Unique Firsthand Account of Formative Experiences with Alan Turing*. Communications of the ACM 55(12):39-41, 2012.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV





Kap. 10

Kap. 11




Teil V

Kap. 12

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 12 (8)

-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 9, Formalismen 1: Zur Semantik von Funktionen)
-  Rózsa Péter. *Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktionen*. Mathematische Annalen 110:612-632, 1934.
-  Rózsa Péter. *Konstruktion nichtrekursiver Funktionen*. Mathematische Annalen 111:42-60, 1935.
-  Gordon Plotkin. *Call-by-name, Call-by-value, and the  $\lambda$ -Calculus*. Theoretical Computer Science 1:125-159, 1975.

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 12 (9)

-  Uwe Schöning, Wolfgang Thomas. *Turings Arbeiten über Berechenbarkeit – eine Einführung und Lesehilfe*. Informatik Spektrum 35(4):253-260, 2012. (Abschnitt Äquivalenz zwischen Turingmaschinen und Lambda-Kalkül)
-  Boris A. Trakhtenbrot *Comparing the Church and Turing Approaches: Two Prophetic Messages*. In Rolf Herken (Hrsg.), *The Universal Turing Machine: A Half-Century Survey*. Springer-V., 2. Auflage, 557-582, 1995.
-  Allen B. Tucker (Editor-in-Chief). *Computer Science Handbook*. Chapman & Hall/CRC, 2004.  
(Kapitel 92.3, The Lambda Calculus: Foundation of All Functional Languages)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 12 (10)



Ingo Wegener. *Grenzen der Berechenbarkeit*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 111-118, 2006. (Kapitel 4.1, Rechnermodelle und die Churchsche These)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Kapitel 13

## Auswertungsordnungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Kapitel 13.1

## Motivation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Motivation und Überblick

Zwei **Hauptauswertungsordnungen**:

- ▶ **Applikative** Auswertung
  - ▶ **Kennzeichnend**: Sofortige, unverzügliche Argumentauswertung in Funktionsaufrufen.
- ▶ **Normale** Auswertung
  - ▶ **Kennzeichnend**: Aufgeschobene, verzögerte Argumentauswertung in Funktionsaufrufen.

**Operationalisierungen** applikativer u. normaler Auswertung als

- ▶ **linksapplikative**, **frühe** (oder **fleißige**) Auswertung (engl. **eager evaluation**).
- ▶ **linksnormale** Auswertung (ohne implementierungspraktische Bedeutung).
- ▶ **linksnormale** Auswertung mit **Ausdrucksteilung**, **späte** (oder **faule**) Auswertung (engl. **lazy evaluation**).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# 'Aliase' applikativer und normaler Auswertung

## Applikative Auswertungsordnung (engl. applicative order eval.)

- ▶ **Verwandte Bezeichnungen:** Wertparameter-, innerste oder strikte Auswertung (engl. call-by-value, innermost or strict evaluation).
- ▶ **Operationalisierung:** Linksapplikative, linkestinnerste, **frühe** (oder **fleißige**) **Auswertung** (engl. leftmost- innermost or **eager evaluation**).

## Normale Auswertungsordnung (engl. normal order evaluation)

- ▶ **Verwandte Bezeichnungen:** Namensparameter-, äußerste Auswertung (engl. call-by-name, outermost evaluation).
- ▶ **Operationalisierung:** Linksnormale, linkestäußerste Auswertung (engl. leftmost-outermost evaluation).
- ▶ **Effiziente Operationalisierung mit Ausdrucksteilung:** **Späte** (oder **faule**) **Auswertung** (engl. **lazy evaluation**).
  - ▶ **Verwandte Bezeichnung:** Bedarfsparameter-Auswertung (engl. call-by-need evaluation).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 13.2

## Applikative, normale Auswertungsordnung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Argumentauswertung in Funktionsaufrufen (1)

Applikativ: **Sofortige, unverzügliche** Argumentauswertung.

- ▶ Ein applikativer Ausdruck  $(f \text{ ausd}_1 \dots \text{ausd}_n)$  wird ausgewertet, indem
  - ▶ zunächst die Argumentausdrücke  $\text{ausd}_1, \dots, \text{ausd}_n$  vollständig ausgewertet werden, anschließend ihre Werte  $w_1, \dots, w_n$  im Rumpf von  $f$  für die Parameter von  $f$  eingesetzt werden und schließlich der so entstandene expandierte Ausdruck ausgewertet wird.

Die Wirkung **sofortiger** Argumentauswertung motiviert folgende **Bezeichnungsvarianten**:

- ▶ Wertparameter-, innerste, strikte Auswertung (engl. *call-by-value evaluation*, *innermost evaluation*, *strict evaluation*).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Argumentauswertung in Funktionsaufrufen (2)

Normal: **Aufgeschobene**, **verzögerte** Argumentauswertung.

- ▶ Ein applikativer Ausdruck  $(f \text{ ausd}_1 \dots \text{ausd}_n)$  wird ausgewertet, indem
  - ▶ die Argumentausdrücke  $\text{ausd}_1, \dots, \text{ausd}_n$  unausgewertet im Rumpf von  $f$  für die Parameter von  $f$  eingesetzt werden und anschließend der so entstandene expandierte Ausdruck ausgewertet wird.

Die Wirkung **aufgeschobener** Argumentauswertung motiviert folgende **Bezeichnungsvarianten**:

- ▶ Namensparameter-, äußerste Auswertung (engl. **call-by-name evaluation**, **outermost evaluation**).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Grundthema aller Auswertungsstrategien

...die **Organisation** des Zusammenspiels von

- ▶ **Expandieren** (von Funktionsaufrufen)
- ▶ **Simplifizieren** (von Ausdrücken verschieden von Funktionsaufrufen)

mit dem Ziel, Ausdrücke **so weit** zu vereinfachen **wie möglich**, ihren **Wert** zu **berechnen**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Drei Beispiele zur Illustration

## 1. Arithmetischer Ausdruck:

`3 * (9+5) ->> ...`

## 2. Ausdruck mit nichtrekursivem Funktionsaufruf und a) elementaren, b) komplexen Argumentausdrücken:

`simple :: Int -> Int -> Int -> Int`

`simple x y z = (x+z) * (y+z)`

`simple 2 3 4 ->> ...`

## 3. Ausdruck mit rekursivem Funktionsaufruf:

`fac :: Integer -> Integer`

`fac n = if n == 0 then 1 else n * fac (n-1)`

`fac 2 ->> ...`



# Bsp. 1: Arithmetischer Ausdruck

Viele **Simplifikations (S)-Wege** – alle führen zum selben Wert:

S-Weg 1:

$$\begin{aligned} & 3 * (9+5) \\ (S) \rightarrow & 3 * 14 \\ (S) \rightarrow & 42 \end{aligned}$$

S-Weg 2:

$$\begin{aligned} & 3 * (9+5) \\ (S) \rightarrow & 3*9 + 3*5 \\ (S) \rightarrow & 27 + 3*5 \\ (S) \rightarrow & 27 + 15 \\ (S) \rightarrow & 42 \end{aligned}$$

S-Weg 3:

$$\begin{aligned} & 3 * (9+5) \\ (S) \rightarrow & 3*9 + 3*5 \\ (S) \rightarrow & 3*9 + 15 \\ (S) \rightarrow & 27 + 15 \\ (S) \rightarrow & 42 \end{aligned}$$

## Bsp. 2a): Aufruf mit elementaren Argumenten

```
simple x y z :: Int -> Int -> Int -> Int  
simple x y z = (x + z) * (y + z)
```

Eine **Expansion (E)**, viele **Simplifikationen (S)** – alle Wege führen zum selben Wert:

ES-Weg 1:

```
simple 2 3 4  
(E) ->> (2 + 4) * (3 + 4)  
(S) ->> 6 * (3 + 4)  
(S) ->> 6 * 7  
(S) ->> 42
```

ES-Weg 2:

```
simple 2 3 4  
(E) ->> (2 + 4) * (3 + 4)  
(S) ->> (2 + 4) * 7  
(S) ->> 6 * 7  
(S) ->> 42
```

ES-Weg 3:

```
simple 2 3 4  
(E) ->> ... ->> 42
```

## Bsp. 2b): Aufruf mit komplexen Argumenten

```
simple x y z :: Int -> Int -> Int -> Int  
simple x y z = (x + z) * (y + z)
```

### ES-Weg 1: Applikative Auswertung

```
simple 2 3 ((5+7)*9)    (Arg.vereinf. zuerst)  
(S) ->> simple 2 3 12*9  
(S) ->> simple 2 3 108  
(E) ->> (2 + 108) * (3 + 108)  
(S) ->> ...  
(S) ->> 12.210
```

### ES-Weg 2: Normale Auswertung

```
simple 2 3 ((5+7)*9)    (Expansion zuerst)  
(E) ->> (2 + ((5+7)*9)) * ((3 + (5+7)*9))  
(S) ->> ...  
(S) ->> 12.210
```

## Beispiel 3: Rekursiver Aufruf

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

fac 2

(E) ->> if 2 == 0 then 1 else (2 \* fac (2 - 1))

(S) ->> if False then 1 else (2 \* fac (2 - 1))

(S) ->> 2 \* (fac (2 - 1))

Auch hier gibt es die Möglichkeit

- ▶ **applikativ** (d.h., rechnen auf Argumentposition)
- ▶ **normal** (d.h., Aufruf expandieren)

auswertend fortzufahren.

...wir führen beide Möglichkeiten im Detail aus.

### Bsp. 3: Applikative vs. normale Auswertung

Applikativ:  $2 * \text{fac } (2-1)$  (Arg.vereinf. zuerst)

(S)  $\rightarrow$  2 \* fac 1

```
(E) ->> 2 * (if 1 == 0 then 1 else (1 * fac (1-1)))
```

```
(S) ->> 2 * (if False then 1 else (1 * fac (1-1)))
```

(S)  $\rightarrow 2 * (1 * \text{fac } (1-1))$

(S) ->> ...in diesem Stil fortfahren.

Normal:  $2 * \text{fac}(2-1)$  (Expansion zuerst)

```
(E) ->> 2 * (if (2-1) == 0 then 1
                else ((2-1) * fac ((2-1)-1)))
```

```
(S) ->> 2 * (if 1 == 0 then 1
              else ((2-1) * fac ((2-1)-1)))
```

```
(S) ->> 2 * (if False then 1
              else ((2-1) * fac ((2-1)-1)))
```

(S)  $\rightarrow 2 * ((2-1) * \text{fac}((2-1)-1))$

(S)  $\rightarrow 2 * (1 * \text{fac}((2-1)-1))$

(E) ->> ...in diesem Stil fortfahren.

## Bsp. 3: Applikative Auswertung (1)

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

```
    fac (1+1)           (Argument wird sofort ausgewertet)
```

```
(S) ->> fac 2           (Expansion nach max. Argumentvereinf.)
```

```
(E) ->> if 2 == 0 then 1 else (2 * fac (2-1))
```

```
(S) ->> if False then 1 else (2 * fac (2-1))
```

```
(S) ->> 2 * fac (2-1)     (Arg. wird sofort ausgewertet)
```

```
(S) ->> 2 * fac 1         (Exp. nach max. Argumentvereinf.)
```

```
(E) ->> 2 * (if 1 == 0 then 1  
             else (1 * fac (1-1)))
```

```
(S) ->> 2 * (if False then 1  
             else (1 * fac (1-1)))
```

```
(S) ->> 2 * (1 * fac (1-1)) (Arg. wird sofort ausgewertet)
```

```
(S) ->> 2 * (1 * fac 0)     (Exp. nach max. Arg.vereinf.)
```

```
(E) ->> 2 * (1 * (if 0 == 0 then 1  
                  else (0 * fac (0-1))))
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## Bsp. 3: Applikative Auswertung (2)

```
(S) ->> 2 * (1 * (if False then 1  
                  else (0 * fac (0-1))))
```

```
(S) ->> 2 * (1 * 1)
```

```
(S) ->> 2 * 1
```

```
(S) ->> 2
```

~> Applikative, sofortige Argumentauswertung  
(engl. applicative order evaluation)

## Bsp. 3: Normale Auswertung (1)

fac n = if n == 0 then 1 else (n \* fac (n - 1))

fac (1+1) (Sofortige Exp., keine vorh. Arg.vereinf.)

(E) ->> if (1+1) == 0 then 1  
          else ((1+1) \* fac ((1+1)-1))

(S) ->> if 2 == 0 then 1  
          else ((1+1) \* fac ((1+1)-1))

(S) ->> if False then 1  
          else ((1+1) \* fac ((1+1)-1))

(S) ->> ((1+1) \* fac ((1+1)-1))

(S) ->> (2 \* fac ((1+1)-1)) (Sofortige Exp.)

(E) ->> 2 \* (if ((1+1)-1) == 0 then 1  
              else (((1+1)-1) \* fac (((1+1)-1)-1)))

(3S) ->> 2 \* (if False then 1  
              else (((1+1)-1) \* fac (((1+1)-1)-1)))

(S) ->> 2 \* (((1+1)-1) \* fac (((1+1)-1)-1))

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



## Bsp. 3: Normale Auswertung (2)

(S) ->> 2 \* ((2-1) \* fac (((1+1)-1)-1))

(S) ->> 2 \* (1 \* **fac** (((1+1)-1)-1)) - Sofortige Exp.

(E) ->> 2 \* (1 \* (if (((1+1)-1)-1) == 0 then 1  
else (((1+1)-1)-1) \* fac (((1+1)-1)-1))))

(4S) ->> 2 \* (1 \* (if **True** then 1  
else (((1+1)-1)-1) \* fac (((1+1)-1)-1))))

(S) ->> 2 \* (1 \* 1)

(S) ->> 2 \* 1

(S) ->> 2

⇝ Normale, aufgeschobene Argumentauswertung  
(engl. normal order evaluation)

# Kapitel 13.3

## Linksapplikative, linksnormale Auswertungs- ordnung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Linksapplikative, linksnormale

...Auswertungsordnungen als wichtige praktische **Operationalisierungen**

- ▶ applikativer
- ▶ normaler

Auswertung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Applikativ oder normal auszuwerten

...beantwortet eine erste **operationell wichtige** Frage der Ausdrucksauswertung:

## 1. Wie ist mit (Funktions-) Argumenten umzugehen?

↪ Ausgewertet oder unausgewertet übergeben?

- ▶ **Applikativ** (**innerst**): **Ausgewertet** übergeben.
- ▶ **Normal** (**äußerst**): **Unausgewertet** übergeben.

**Linksapplikativ** oder **linksnormal** auszuwerten beantwortet eine zweite **operationell wichtige** Frage:

## 2. Wo ist im Ausdruck auszuwerten?

↪ Links, rechts, halblinks, in der Mitte?

- ▶ **Linksapplikativ** (**linksinnerst**): **Linkstinnerste** Stelle.
- ▶ **Linksnormal** (**linksäußerst**): **Linkstäußerste** Stelle.

$2^3 + \text{fac}(\text{fib}(\text{squ}(2+2))) + 3*5 + \text{fib}(\text{fac}(7*(5+3))) + \text{fib}((5+7)*2)$

# Linksapplikative Auswertung

Linksapplikativ ausgewertet:

```
2^3+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib(squ 4))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib((4*4)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib 16)+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> 8+fac(fib (16-2)+fib(16-1)) + 3*5 +...
->> 8+fac(fib 14+fib(16-1)) + 3*5 +...
->> ...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Linksnormale Auswertung

Linksnormal ausgewertet:

```
2^3+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
```

```
->> 8+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
```

```
->> 8+(if fib(squ(2+2))==0 then 1 else n*fac(fib(squ(2+2))-1))
      +3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
```

```
->> 8+(if (if squ(2+2)==0 then 0
           else if squ(2+2)==1 then 1
                else fib(squ(2+2)-2)+fib(squ(2+2)-1)) ==0
      then 1 else n*fac(fib(squ(2+2))-1)) + 3*5 +...
```

```
->> 8+(if (if (2+2)*(2+2)==0 then 0
           else if squ(2+2)==1 then 1
                else fib(squ(2+2)-2)+fib(squ(2+2)-1)) ==0
      then 1 else n*fac(fib(squ(2+2))-1)) + 3*5 +...
```

```
->> 8+(if (if 4*(2+2)==0 then 0
           else if squ(2+2)==1 then 1
                else fib(squ(2+2)-2)+fib(squ(2+2)-1)) ==0
      then 1 else n*fac(fib(squ(2+2))-1)) + 3*5 +...
```

```
->> ...
```

Inhalt

Teil I

1

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Beachte bei der Expansion von fib

...im Fall der **linksnormalen** Auswertung:

Die **musterbasierte** Definition von **fib**:

```
fib :: Int -> Int      -- Musterbasierte Definition
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

ist **syntaktischer Zucker** der Grundversion mit **geschachteltem Fallunterscheidungsausdruck**:

```
fib :: Int -> Int      -- Grundversion
fib n = if n==0 then 0
        else if n==1 then 1
              else fib (n-2) + fib (n-1)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Zentrale Fragen

Welche Auswirkungen hat die Wahl von (links-) applikativer oder (links-) normaler Auswertungsordnung?

...auf

- ▶ Terminierungsverh., -häufigk.? (Th. 13.3.2, Th. 13.3.3)
- ▶ Terminierungsgeschwindigkeit? (Th. 13.3.3, Kap. 13.5)
- ▶ berechneten Ausdruckswert im Term.fall? (Th. 13.3.1)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Hauptresultate (1)

Die Church/Rosser-Theoreme 12.3.4.1, 12.3.4.2 garantieren:

## Theorem 13.3.1 (Wertdeterminiertheit)

Jede terminierende Auswertungsfolge endet mit demselben Ergebnis.

**Informell:** Terminierende Auswertungsfolgen widersprechen sich nicht.

## Theorem 13.3.2 (Terminierungshäufigkeit)

Wenn irgendeine Auswertungsfolge terminiert, so terminiert auch die (links-) normale Auswertungsreihenfolge.

**Informell:** (Links-) normale Auswertung terminiert am häufigsten.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Hauptresultate (2)

## Theorem 13.3.3 (Abweichendes Terminierungsverh.)

Angesetzt auf einen Ausdruck, kann die (links-) normale Auswertungsordnung terminieren, die (links-) applikative aber nicht.

Informell: (links-) applikative und (links-) normale Auswertungsordnung können sich abhängig vom Ausdruck unterscheiden in

- ▶ Terminierungsverhalten (d.h. Terminierungshäufigkeit)
- ▶ Terminierungsgeschwindigkeit (d.h. Performanz)

nicht aber hinsichtlich des

- ▶ Ergebnisses, wenn sie terminieren.

# Zu Terminierungsgeschwindigkeit, -häufigkeit

...anhand von Beispielen über den Funktionen `squ`, `infinite` und `fst`:

- ▶ Die Quadratfunktion `squ` auf ganzen Zahlen:

```
squ :: Integer -> Integer  
squ n = n * n
```

- ▶ Die 'ewige' Inkrementfunktion `infinite`:

```
infinite :: Integer  
infinite = 1 + infinite
```

- ▶ Die Projektionsfunktion `fst`:

```
fst :: (a,b) -> a  
fst (x,_) = x
```

- ▶ Drei **Ausdrücke**:

```
(17+4) + squ (squ (squ (1+1))) + (2*11)  
fst (2*21,squ (squ (squ (1+1))))  
fst (2*21,infinite)
```

# Linksapplikative, linkestininnerste (LI) Auswert.

...von  $(17+4) + \text{squ}(\text{squ}(\text{squ}(1+1))) + (2*11)$ :

$((17+4) + \text{squ}(\text{squ}(\text{squ}(1+1)))) + (2*11)$

(LI-S)  $\rightarrow (21 + \text{squ}(\text{squ}(\text{squ}(1+1)))) + (2*11)$

(LI-S)  $\rightarrow (21 + \text{squ}(\text{squ}(\text{squ } 2))) + (2*11)$

(LI-E)  $\rightarrow (21 + \text{squ}(\text{squ}(2*2))) + (2*11)$

(LI-S)  $\rightarrow (21 + \text{squ}(\text{squ } 4)) + (2*11)$

(LI-E)  $\rightarrow (21 + \text{squ}(4*4)) + (2*11)$

(LI-S)  $\rightarrow (21 + \text{squ } 16) + (2*11)$

(LI-E)  $\rightarrow (21 + 16*16) + (2*11)$

(LI-S)  $\rightarrow (21 + 256) + (2*11)$

(LI-S)  $\rightarrow 277 + (2*11)$

(LI-S)  $\rightarrow 277 + 22$

(LI-S)  $\rightarrow 299$

Insgesamt:  $1 + 7 + 3 = 11$  Schritte

...davon 7 Schritte für  $\text{squ}(\text{squ}(\text{squ}(1+1)))$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Linksnormale, linkestäußerste (LÄ) Ausw. (1)

...von  $(17+4) + \text{squ}(\text{squ}(\text{squ}(1+1))) + (2*11)$ :

$((17+4) + \text{squ}(\text{squ}(\text{squ}(1+1)))) + (2*11)$

(LÄ-S)  $\rightarrow (21 + \text{squ}(\text{squ}(\text{squ}(1+1)))) + (2*11)$

(LÄ-E)  $\rightarrow (21 + \text{squ}(\text{squ}(1+1)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-E)  $\rightarrow (21 + ((\text{squ}(1+1)) * (\text{squ}(1+1))) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-E)  $\rightarrow (21 + ((1+1) * (1+1) * \text{squ}(1+1)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow (21 + (2*(1+1) * \text{squ}(1+1)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow (21 + (2*2 * \text{squ}(1+1)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow (21 + (4 * \text{squ}(1+1)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-E)  $\rightarrow (21 + (4*((1+1)*(1+1))) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow (21 + (4*(2*(1+1))) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow (21 + (4*(2*2)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow (21 + (4*4) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow (21 + 16 * \text{squ}(\text{squ}(1+1))) + (2*11)$

$\rightarrow \dots$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Linksnormale, linkestäußerste (LÄ) Ausw. (2)

```
->> ...  
(LÄ-S) ->> (21 + (16 * 16)) + (2*11)  
(LÄ-S) ->> (21 + 256) + (2*11)  
(LÄ-S) ->> 277 + (2*11)  
(LÄ-S) ->> 277 + 22  
(LÄ-S) ->> 299
```

Insgesamt:  $1 + ((1+10)+(1+10)+1) + 3 = 27$  Schritte  
...davon 23 Schritte für `squ (squ (squ (1+1)))`

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Linksapplikativ effizienter als linksnormal?

Nicht immer! Betrachte den zweiten Ausdruck:

```
first (2*21, squ (squ (squ (1+1))))
```

- (Links-) applikative Auswertung:

```
first (2*21, squ (squ (squ (1+1))))
```

```
(LI-S) ->> first (42, squ (squ (squ (1+1))))
```

```
->> ...
```

```
(LI-S) ->> first (42, 256)
```

```
(LI-E) ->> 42
```

Insgesamt:  $1+7+1=9$  Schritte (davon 7 für den Wert des unbenötigten zweiten Arguments!)

- (Links-) normale Auswertung:

```
first (2*21, squ (squ (squ (1+1))))
```

```
(LÄ-E) ->> 2*21
```

```
(LÄ-S) ->> 42
```

Insgesamt: 2 Schritte (das unbenötigte zweite Argument wird überhaupt nicht ausgewertet!)

# Linksapplikativ, linksnormal 'termin.-gleich'?

Nicht immer! Betrachte den dritten Ausdruck:

```
first (2*21,infinite)
```

- (Links-) applikative Auswertung:

```
first (2*21,infinite)
```

```
(LI-S) ->> first (42,infinite)
```

```
(LI-E) ->> first (42,1+infinite)
```

```
(LI-E) ->> first (42,1+(1+infinite))
```

```
(LI-E) ->> ...
```

```
(LI-E) ->> first (42,1+(1+(1+(...+(1+infinite)...)))
```

```
(LI-E) ->> ...
```

Insgesamt: Nichtterminierung, kein Resultat: **undefiniert!**

- (Links-) normale Auswertung:

```
first (2*21,infinite)
```

```
(LÄ-E) ->> 2*21
```

```
(LÄ-S) ->> 42
```

Insgesamt: Terminierung, Res. 42 nach 2 Schritten: **def.!**

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17



# Unterschiedliche Terminierungshäufigkeit

..von linksapplikativer und linksnormaler Auswertung.

Das Beispiel des zweiten Ausdrucks

`first (2*21,squ (squ (squ (1+1))))`

zeigt:

- ▶ (Links-) applikative und (links-) normale Auswertung können sich im Terminierungsverhalten unterscheiden:
  - ▶ Applikativ: Nichttermination, kein Resultat: undefiniert.
  - ▶ Normal: Termination, sehr wohl ein Resultat: definiert.

Bem.: Die umgekehrte Situation ist nicht möglich (s. Theorem 13.3.1)!

# Unterschiedliche Terminierungsgeschwindigkeit

...von linksapplikativer und linksnormaler Auswertung.

Das Beispiel des ersten Ausdrucks

$(17+4) + \text{squ}(\text{squ}(\text{squ}(1+1)))) + (2*11)$

zeigt:

- ▶ **Ergebnisgleichheit:** Terminieren (links-) applikative und (links-) normale Auswertungsordnung angewendet auf einen Ausdruck beide, so terminieren sie mit demselben Resultat (s. Theorem 13.3.1).
- ▶ **Schritzzahlungleichheit:** (Links-) applikative und (links-) normale Auswertung können bis zur Terminierung (mit gleichem Endresultat) unterschiedlich viele Expansions- und Simplifikationsschritte benötigen.

# Unordenbare Performanz

..von **linksapplikativer** und **linksnormaler** Auswertung.

Die Beispiele aller drei **Ausdrücke** zusammen

```
(17+4) + squ (squ (squ (1+1)))) + (2*11)
fst (2*21,squ (squ (squ (1+1))))
fst (2*21,infinite)
```

zeigen, dass weder **linksapplikative** noch **linksnormale** Auswertung der jeweils anderen Auswertungsordnung stets überlegen ist.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# (Links-) normale Auswertung und Effizienz

**Problem:** Naive (links-) normale Auswertung führt häufig zur Mehrfachauswertung von Ausdrücken (siehe etwa **Beispiel 2b**), **Weg 2**, oder die (links-) normale Auswertung des Ausdrucks `squ (squ (squ (1+1)))`.

**Ziel:** Effizienzsteigerung durch Vermeidung von Mehrfachauswertungen von Ausdrücken.

**Methode:** Teilen gemeinsamer Ausdrücke durch Übergang zu Ausdrucksdarstellung in Form von **Graphen**; Ausdrucksauswertungen werden zu **Graphtransformationen**.

**Resultierende Auswertungsordnung:** **Späte** (oder **faule**) Auswertung (engl. **lazy evaluation**)!

...garantiert, dass Argumente **höchstens einmal** ausgewertet werden (möglicherweise also **gar nicht!**).



# Zusammenfassung

## Späte Auswertung (engl. *lazy evaluation*)

- ▶ ist eine *effiziente* Umsetzung (*links-*) normaler Auswertungsordnung.
- ▶ erfordert implementierungstechnisch eine Darstellung von Ausdrücken in Form von Graphen und Graphtransformationen zu ihrer Auswertung.
- ▶ 'vergleichbar' performant wie *frühe Auswertungsordnung* (engl. *eager evaluation*), wenn alle Argumente benötigt werden.
- ▶ vereint möglichst gut die Vorteile applikativer (*Effizienz!*) und normaler (*Terminierungshäufigkeit!*) Auswertungsordnung.

# Kapitel 13.4

## Auswertungsordnungscharakterisierungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Auswertungsordnungscharakterisierungen

...über Analogien und Betrachtungen zu:

- (i) Parameterübergabemechanismen
- (ii) Auswertungspositionen
- (iii) Argumentauswertungshäufigkeiten
- (iv) Definiertheitszusammenhang von Argument und Funktion

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



## (i) Parameterübergabemechanismen analogien

Applikative Auswertungsordnung entspricht

- ▶ Call-by-value

Normale Auswertungsordnung entspricht

- ▶ Call-by-name

Späte Auswertungsordnung entspricht

- ▶ Call-by-need

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## (ii) Auswertungspositionen

### Applikative Auswertungsordnung

- ▶ **Innerste Auswertungsordnung:** Reduziere nur Redexe, die keine Redexe enthalten.
- ▶ **Linksapplikative, linkestinnerste Auswertungsordnung:** Reduziere stets den linken innersten Redex, der keine Redexe enthält.

**Frühe Auswertungsordnung** (engl. **eager evaluation**).

### Normale Auswertungsordnung

- ▶ **Äußerste Auswertungsordnung:** Reduziere nur Redexe, die nicht in anderen Redexen enthalten sind.
- ▶ **Linksnormale, linkestäußerste Auswertungsordnung:** Reduziere stets den linken äußersten Redex, der nicht in anderen Redexen enthalten ist.

**Späte Auswertungsordnung** (engl. **lazy evaluation**), effiziente Umsetzung linksnormaler Auswertung.

## (iii) Argumentauswertungshäufigkeit

### Applikative Auswertungsordnung

- ▶ Jedes Argument wird **genau einmal** ausgewertet.

### Normale Auswertungsordnung

- ▶ Jedes Argument wird **so oft** ausgewertet, **wie es benutzt** wird.

### Späte Auswertungsordnung

- ▶ Jedes Argument wird **höchstens einmal** ausgewertet.

### (iii) Illustrierendes Beispiel

Betrachte die **Funktion**:

```
f :: Int -> Int -> Int -> Int
f x y z = if x>42 then y+y else z^z
```

und den **Aufruf**:

```
f 45 (squ (5*(2+3))) (squ ((2+3)*7))
```

### (iii) Applikative Auswertung von f

`f x y z = if x>42 then y+y else z^z`

Applikative Auswertung:

```
f 45 (squ (5*(2+3))) (squ ((2+3)*7))
(2S) ->> f 45 (squ (5*5)) (squ (5*7))
(2S) ->> f 45 (squ 25) (squ 35)
(2E) ->> f 45 (25*25) (35*35)
(2S) ->> f 45 625 1225
(E) ->> if 45>42 then 625+625 else 1125^1125
(S) ->> if True then 625+625 else 1125^1125
(S) ->> 625+625
(S) ->> 1250
```

...die Argumente `(squ (5*(2+3)))` und `(squ ((2+3)*7))` werden beide **genau einmal** ausgewertet (ohne dass der Wert von `(squ ((2+3)*7))` benötigt wird).

### (iii) Normale Auswertung von f

`f x y z = if x>42 then y+y else z^z`

Normale Auswertung:

```
f 45 (squ (5*(2+3))) (squ ((2+3)*7))
(E) ->> if 45>42 then (squ (5*(2+3))) + (squ (5*(2+3)))
      else (squ ((2+3)*7)) * (squ ((2+3)*7))
(S) ->> if True then (squ (5*(2+3))) + (squ (5*(2+3)))
      else (squ ((2+3)*7))^(squ ((2+3)*7))
(S) ->> (squ (5*(2+3))) + (squ (5*(2+3)))
(2S) ->> (squ (5*5)) + (squ (5*5))
(2S) ->> (squ 25) + (squ 25)
(2E) ->> (25*25) + (25*25)
(2S) ->> 625 + 625
(S) ->> 1250
```

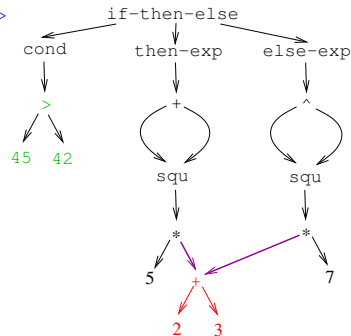
...das Argument `(squ (5*(2+3)))` wird **zweimal** ausgewertet;  
das nicht benötigte Argument `(squ ((2+3)*7))` gar nicht.

### (iii) Späte Auswertung von f

f x y z = if 42>x then y+y else z^z

f 45 (squ (5\*(2+3))) (squ ((2+3)\*7))

(E) ->>



Späte  
Auswertung:

(S) ->> ... (S) ->> 1250

...das Argument (squ (5\*(2+3))) wird genau einmal ausgewertet; vom nicht benötigten Argument (squ ((2+3)\*7)) der Teilterm (2+3) (wg. Ausdrucksteilung ohne Extrakosten!).

## (iv) Definiertheitszusammenhang

...von **Argument** und **Funktion**.

**Schlüsselbegriff:** Striktheit von Funktionen.

### Definition 13.4.1 (Strikt im $n$ -ten Parameter)

Eine Funktion  $f$  heißt **strikt** in ihrem  $n$ -ten Parameter (oder **Argument**), wenn gilt: Ist der Wert des Arguments des  $n$ -ten Parameters nicht definiert, so ist auch der Wert von  $f$  nicht definiert (unabhängig von den Werten möglicher weiterer Argumente).



## (iv) Bsp.: Striktheit bei einstelligen Funktionen

Die Fakultäts- und Fibonacci-Funktion sind **strikt** in ihrem ersten (und einzigen) Parameter. Undefiniertheit des Argumentwerts impliziert Undefiniertheit der Funktion.

```
fac (1 'div' 0) (LI-S) ->> undef
```

```
fac (1 'div' 0) (LÄ-E) ->>
```

```
  if ((1 'div' 0) == 0) then 1
```

```
    else n * fac ((1 'div' 0) - 1) (LÄ-S) ->> undef
```

```
fib (1 'div' 0) (LI-S) ->> undef
```

```
fib (1 'div' 0) (LÄ-E) ->>
```

```
  if (1 'div' 0) == 0 then 0
```

```
    else (1 'div' 0) == 1 then 1
```

```
      else fib ((1 'div' 0) - 2) + fib ((1 'div' 0) - 1)
```

```
      (LÄ-S) ->> undef
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17

## (iv) Bsp.: Striktheit bei mehrstelligen Fkt.

Mehrstellige Funktionen können **strikt** in einigen Parametern, **nicht strikt** in anderen sein:

Der **Fallunterscheidungsausdruck** (-funktion)

`(if . then . else .)`

ist **strikt** im 1-ten Argument (Bedingung), **nicht strikt** im 2-ten und 3-ten Argument (then- und else-Ausdruck).

```
if ((1 'div' 0) == 0) then 4 else 2 ->> undef
```

(strikt in Bedingung)

```
if ((0 'div' 1) == 0) then 42 else 1 'div' 0
->> if (0 == 0) then 42 else 1 'div' 0
->> if True then 42 else 1 'div' 0
->> 42
```

(nicht strikt im 3-ten Arg.)

```
if ((0 'div' 1) /= 0) then 1 'div' 0 else 42
->> if (0 /= 0) then 1 'div' 0 else 42
->> if False then 1 'div' 0 else 42
->> 42
```

(nicht strikt im 2-ten Arg.)

## (iv) Striktheit, Terminierung, Ergebnisneutralit.

### Theorem 13.4.2 (Striktheit, Terminierung)

Für strikte Funktionen stimmen die Terminierungsverhalten von früher und später Auswertungsordnung für die strikten Argumente überein.

### Korollar 13.4.3 (Striktheit, Ergebnisneutralität)

Durch den Übergang von später auf frühe Auswertung für strikte Argumente einer Funktion gehen keine Ergebnisse verloren (und stimmen gemäß Theorem 13.3.1 überein).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Striktheit und Optimierung

Für strikte Argumente von Funktionen darf deshalb stets

- späte durch frühe Auswertung ersetzt werden

da sich Terminierungsverhalten und Resultat nicht ändern.

Die Ersetzung später durch frühe Auswertung für strikte Argumente von Funktionen ist eine der wichtigsten

- Optimierungen

bei der Übersetzung funktionaler Sprachen mit später Auswertungssemantik.

**Beispiel:** Frühe statt späte Argumentauswertung für die in ihrem jeweiligen Argument strikten Fakultäts- und Fibonacci-Funktionen ist für Übersetzer von Sprachen wie Haskell und Miranda erlaubt.

# Striktheitsanalyse, strikte Auswertung

Übersetzer spät auswertender Sprachen führen dazu eine sog.

- ▶ Striktheitsanalyse

durch, um dort, wo es **sicher** ist, d.h. wo ein Ausdruck zum Ergebnis beiträgt und sein Wert deshalb in **jeder** Auswertungsordnung benötigt wird,

- ▶ **aufgeschobene, späte** (engl. **lazy**)

durch

- ▶ **sofortige, frühe** (engl. **eager**)

**Argumentauswertung** zu ersetzen.

Statt von **früher** Auswertung spricht man deshalb auch von

- ▶ **striktter Auswertung** (engl. **strict evaluation**).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 13.5

## Frühe oder späte Auswertung? Eine Standpunktfrage

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

To be, or not to be,  
that is the question.

*Hamlet, Prinz von Dänemark*  
William Shakespeare (1564-1616)  
engl. Dramatiker und Lyriker

To be **lazy**, or not to be **lazy**,  
that is the question.

*...Hamlet, zeitgerecht  
interpretiert*

- ▶ **Frühe Auswertung** von Argumenten (engl. **eager evaluation**) (in Sprachen wie **ML**, **Scheme** (ohne Makros),...)
- ▶ **Späte Auswertung** von Argumenten (engl. **lazy evaluation**) (in Sprachen wie **Haskell**, **Miranda**,...)

Quot capita, tot sententiae.  
Wie viele Köpfe, so viele Ansichten.

Terenz (190 v.Chr. - 159 v.Chr.)  
röm. Schriftsteller

# Späte vs. frühe Argumentauswertung (1)

...die Vorteile des einen sind die Nachteile des anderen und umgekehrt.

**Vorteile später Argumentauswertung** (mit Ausdrucksteilung):

- ▶ Terminiert mit Normalform, wenn es (irgend-) eine terminierende Auswertungsreihenfolge gibt.  
**Informell:** Späte (wie normale und linksnormale) Auswertungsordnung terminieren häufigst möglich!
- ▶ Wertet Argumente nur aus, wenn deren Werte wirklich benötigt werden; und dann nur einmal.
- ▶ Ermöglicht eleganten und flexiblen Umgang mit potentiell unendlichen Werten von Datenstrukturen (z.B. unendliche Listen, Ströme (s. [Kap. 18.2](#)), unendliche Bäume, etc.).



# Späte vs. frühe Argumentauswertung (2)

## Nachteile später Argumentauswertung:

- ▶ Konzeptuell und implementierungstechnisch anspruchsvoller.
  - ▶ Repräsentation von Ausdrücken in Form von Graphen statt linearer Sequenzen; Ausdrucksauswertung und -manipulation als Graph- statt Sequenzmanipulation.
  - ▶ Partielle Auswertung von Ausdrücken kann Seiteneffekte bewirken! (Beachte: Einwand gilt nicht für Haskell; in Haskell keine Seiteneffekte! In Scheme: Seiteneffektvermeidung obliegt dem Programmierer.)
  - ▶ Ein-/Ausgabe nicht in trivialer Weise transparent für den Programmierer zu integrieren.

...volle Einsicht in die Nachteilsursachen erfordert tiefergehendes Verständnis von  $\lambda$ -Kalkül und Bereichstheorie (engl. domain theory).

# Späte vs. frühe Argumentauswertung (3)

## Vorteile früher Argumentauswertung:

- ▶ Konzeptuell und implementierungstechnisch einfacher.
- ▶ Einfache(re) Integration imperativer Konzepte.
- ▶ Vom mathematischen Standpunkt oft 'natürlicher'.

Beispiel: Soll der Wert von Ausdrücken wie `(first (2*21,infinite))` definiert gleich 42 sein wie bei aufgeschobener Auswertung oder undefiniert wg. Nichtterminierung wie bei sofortiger Auswertung?

`first (2*21,infinite)) ->> 2*21 ->> 42`

aufgeschobene  
Argumentauswertung

`first (2*21,infinite))`  
`->> first(42,1+infinite)`

sofortige `->> first(42,1+(1+infinite)) ->> ...`

Argumentauswertung

# Auswertungsordnungsauswahlhilfe (1)

... auf Grundlage der Anzahl von Argumentsauswertungen.

## Normale Auswertungsordnung

- ▶ Argumente werden **so oft** ausgewertet, **wie** sie **verwendet** werden.
  - + Kein Argument wird ausgewertet, dessen Wert nicht benötigt wird.
  - + Terminiert, wenn immer es eine terminierende Auswertungsfolge gibt; terminiert am häufigsten, häufiger als applikative Auswertung.
  - Argumente, die mehrfach verwendet werden, werden auch mehrfach ausgewertet; so oft, wie sie verwendet werden  $\rightsquigarrow$  **implementierungspraktisch deshalb irrelevant**; implementierungspraktisch relevant: **aufgeschobene** Auswertung.
- ▶ Theorie-relevant, implementierungspraktisch nicht relevant.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Auswertungsordnungsauswahlhilfe (2)

## Applikative Auswertungsordnung

- ▶ Argumente werden **genau einmal** ausgewertet.
  - + Jedes Argument wird exakt einmal ausgewertet; kein zusätzlicher Aufwand über die Auswertung hinaus.
  - Auch Argumente, deren Wert nicht benötigt wird, werden ausgewertet; das ist kritisch für Argumente, deren Auswertung teuer ist, gar nicht terminiert (unendlich teuer!) oder auf einen Laufzeitfehler führt.

## Späte Auswertungsordnung (mit Ausdrucksteilung)

- ▶ Argumente werden **höchstens einmal** ausgewertet.
  - + Ein Argument wird nur ausgewertet, wenn sein Wert benötigt wird; und dann exakt einmal.
  - + Kombiniert die Vorteile von applikativer Auswertung (**Effizienz!**) und normaler Auswertung (**Terminierung!**).
  - Erfordert zusätzlichen Aufwand zur Laufzeit für die Verwaltung der Auswertung von (Teil-) Ausdrücken.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Auswertungsordnungsauswahlhilfe (3)

...von pragmatischem Standpunkt aus:

- ▶ **Applikative, frühe** Auswertungsordnung vorteilhaft gegenüber (normaler und) später Auswertungsordnung, da
  - ▶ geringere Laufzeitzusatzkosten (engl. overhead).
  - ▶ größeres Parallelisierungspotential (für Funktionsargumente).
- ▶ **Späte** Auswertungsordnung vorteilhaft gegenüber applikativer, früher Auswertungsordnung, wenn
  - ▶ Terminierungshäufigkeit (Definiertheit des Programms!) von überragender Bedeutung.
  - ▶ Argumente nicht benötigt (und deshalb gar nicht ausgewertet) werden  
Bsp.:  $(\lambda x. \lambda y. y) ((\lambda x. x x)(\lambda x. x x)) z \rightarrow (\lambda y. y) z \rightarrow z$
- ▶ **Ideal: Das Beste beider Welten:**
  - ▶ Applikativ, früh, wo möglich; spät, wo nötig.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Auswertungsordnungsauswahlhilfe (4)

...zusammenfassend:

Sofortige, frühe (engl. **eager**) oder aufgeschobene, späte (engl. **lazy**) Argumentauswertung:

- Für **beide** Strategien sprechen **gewichtige** Gründe – und Fürsprecher:

lucundi acti labores.  
Angenehm sind die erledigten Arbeiten.

Cicero (106 - 43 v.Chr.)  
röm. Staatsmann und Schriftsteller

Jetzt schau ma amoi, nacha sehn ma scho!

Franz Beckenbauer (\* 1945)  
bayer. Fußballspieler und Kaiser

- Die Wahl ist letztlich eine Frage von **Angemessenheit** und **Zweckmäßigkeit** im Anwendungskontext.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Randnotiz: 'Lazy' – Maßeinheit

...für den Abstand von Hochkultur und Dekadenz!

To be, or not to be?

To be lazy, or not to be lazy?

...eine Frage der Hochkultur.

...eine Frage der Dekadenz.

O tempora! O mores!

Oh Zeiten! Oh Sitten!

Cato der Ältere (234 - 149 v.Chr.)

röm. Staatsmann

durch Ciceros Reden gegen Verres  
berühmt gewordener Ausruf

Jedoch:

Quae fuerant vitia, mores sunt.

Was früher Fehler waren, sind jetzt die Sitten.

Seneca d. Jüngere (um 4 v.Chr. - 65 n.Chr.)

röm. Politiker, Philosoph und Schriftsteller

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 13.6

## Frühe und späte Auswertung in Haskell

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Steuerung der Auswertung in Haskell

Haskell erlaubt, die Auswertungsordnung (zu einem gewissen Grad) zu steuern.

Aufgeschobene Auswertung:

- Standardverfahren (vom Programmierer nichts zu tun):

```
      fac (2*(3+5))  
(E) ->> if (2*(3+5)) == 0 then 1  
          else ((2*(3+5)) * fac ((2*(3+5))-1))  
...
```

Sofort(-art)ige Auswertung:

- Erzwingbar mithilfe des zweistelligen Operators (\$!):

```
      fac $! (2*(3+5))  
(S) ->> fac $! (2*8)  
(S) ->> fac $! 16  
(E) ->> if 16 == 0 then 1 else (16 * fac $! (16-1))  
...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Sofort(-art)ige Auswertung in Haskell (1)

Wirkung des Operators (`$!`):

- ▶ Die Auswertung des Ausdrucks `(f $! ausd)` erfolgt in gleicher Weise wie die des Ausdrucks `(f ausd)` mit dem Unterschied, dass die Auswertung von `ausd` erzwungen wird, bevor `f` angewendet und expandiert wird.

Im **Detail**: Ist der Wert von `ausd` von einem

- ▶ **elementaren Typ** wie `Int`, `Bool`, `Double`, etc., so wird `ausd` vollständig ausgewertet.
- ▶ **Tupeltyp** wie `(Int,Bool)`, `(Int,Bool,Double)`, etc., so wird `ausd` bis zu einem Tupel von Ausdrücken ausgewertet, aber nicht weiter.
- ▶ **Listentyp**, so wird `ausd` so weit ausgewertet, bis als Ausdruck die leere Liste erscheint oder die Konstruktion zweier Ausdrücke zu einer Liste.

# Sofort(-art)ige Auswertung in Haskell (2)

Für **curryfizierte** Funktionen kann

- ▶ **strikte** Auswertung

für **jede Argumentkombination** erreicht werden.

**Beispiel:** Für die zweistellige curryfizierte Funktion

$$f :: a \rightarrow b \rightarrow c$$

erzwingt

- ▶  $(f \text{ \$! } x) \ y$  : Auswertung von  $x$
- ▶  $(f \ x) \text{ \$! } y$  : Auswertung von  $y$
- ▶  $(f \text{ \$! } x) \text{ \$! } y$  : Auswertung von  $x$  und  $y$

vor der Anwendung und **Expansion** von  $f$ .

# Hauptanwendung von (\$) in Haskell

...zur Minderung des Speicherverbrauchs.

Beispiel: Vergleiche Funktion

```
sumwith_lz :: Int -> [Int] -> Int
sumwith_lz v []          = v
sumwith_lz v (x:xs) = sumwith_lz (v+x) xs
```

mit

```
sumwith_ea :: Int -> [Int] -> Int
sumwith_ea v []          = v
sumwith_ea v (x:xs) = (sumwith_ea ($) (v+x)) xs
```

# Anwendungsbsp.: Aufgeschobene Auswertung

...liefert:

```
sumwith_lz 36 [1,2,3]
(LÄ-E) ->> sumwith_lz (36+1) [2,3,]
(LÄ-E) ->> sumwith_lz ((36+1)+2) [3]
(LÄ-E) ->> sumwith_lz (((36+1)+2)+3) []
(LÄ-E) ->> (((36+1)+2)+3)
(LÄ-S) ->> ((37+2)+3)
(LÄ-S) ->> (39+3)
(LÄ-S) ->> 42
```

...7 Schritte; die max. Länge des summativen Terms auf erster Argumentposition hängt von der Länge der Liste auf zweiter Argumentposition ab.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Anwendungsbsp.: Sofort(-art)ige Auswertung

...mittels (\$) liefert:

```
sumwith_ea 36 [1,2,3]
(LÄ-E) ->> (sumwith_ea $! (36+1)) [2,3]
(LI-S) ->> (sumwith_ea $! 37) [2,3]
(LI-S) ->> sumwith_ea 37 [2,3]
(LÄ-E) ->> (sumwith_ea $! (37+2)) [3]
(LI-S) ->> (sumwith_ea $! 39) [3]
(LI-S) ->> sumwith_ea 39 [3]
(LÄ-E) ->> (sumwith_ea $! (39+3)) []
(LI-S) ->> (sumwith_ea $! 42) []
(LI-S) ->> sumwith_ea 42 []
(LÄ-E) ->> 42
```

...10 Schritte, aber die Länge des summativen Terms auf erster Argumentposition ist unabhängig von der Länge der Liste auf zweiter Argumentposition und konstant kurz.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Anwendungsbsp.: Auswertungsstile

...im Vergleich:

- ▶ Aufgeschobene Auswertung von `sumwith_lz 36 [1..3]`
  - ▶ baut den Ausdruck  $((36+1)+2)+3$  vollständig auf, bevor die erste Simplifikation ausgeführt wird.
  - ▶ Allgemein: `sumwith_lz` baut einen Ausdruck auf, dessen Größe proportional zur Länge der Argumentliste ist.
  - ▶ Problem: Programmabbrüche durch Speicherüberläufe schon für vergleichsweise kleine Argumente möglich:  
`sumwith_lz 36 [1..10000]`
- ▶ Sofort(-art)ige Auswertung von `sumwith_ea 36 [1..3]`
  - ▶ Simplifikationen werden frühestmöglich ausgeführt.
  - ▶ Exzessiver Speicherverbrauch (engl. memory leak) wird dadurch (in diesem Beispiel) vollständig vermieden.
  - ▶ Aber: Die Zahl der Rechenschritte steigt; besseres Speicherverhalten wird gegen schlechtere Schrittzahl eingetauscht (engl. trade-off).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Zusammenfassung

Haskells ( $\$!$ )-Operator ist

- ▶ hilfreich und nützlich

das Speicherverhalten von Programmen zu verbessern, stellt allerdings keinen Königsweg dar: Auch kleine Beispiele erfordern bereits eine

- ▶ sorgfältige Untersuchung

des Verhaltens aufgeschobener und sofort(-art)iger Auswertung.

Es gibt keinen Königsweg [zur Geometrie].

Euklid (ungef. 3./4. Jhdt. v.Chr)  
griech. Mathematiker

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Kapitel 13.7

## Namens- und Bezeichnungsbetrachtung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Englisch vs. Deutsch als Sprachmedium

Englisch	Deutsch	Konnotation
Eager evaluation	Fleißige Auswertung	Positiv
	Eifrige Auswertung	Positiv
	Sofortige Auswertung	Positiv
	Unverzügliche Auswertung	Positiv
	Vorrats-Auswertung	Negativ
	Streber-Auswertung	Negativ
	Frühe Auswertung	Neutral
Lazy evaluation	Faule Auswertung	Negativ
	Lässige Auswertung	Positiv
	Entspannte Auswertung	Positiv
	Aufgeschobene Auswertung	Negativ
	Verzögerte Auswertung	Positiv
	Prokastinator-Auswertung	Negativ
	Auf-den-Punkt-Auswertung	Positiv
	Späte Auswertung	Neutral

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Englische vs. deutsche Begriffspaare

Englisch	Deutsch	Motivation, Bewertung
Eager/lazy	Fleißig/faul	Bestmögl. wörtl.
	Sofortig/aufgeschoben	Neutral, unverfängl.
	Unverzüglich/verzögert	Neutral, unverfängl.
	Sofortig/lässig	Unzusammenpassend,
	Unverzüglich/lässig	keine Gegensatzpaare
	Hektisch/entspannt	Fig. passend, Gegensatzp.
	Angespannt/entspannt	Fig. passend, Gegensatzp.
	Streber/Prokastinator	Mutig, kombiniert Althergebrachtes u. Modewort
	Früh/spät	Neutral, unverfängl.

...jede Wahl vermittelt sprachabhängig 'zwischen den Zeilen' ein bestimmtes **Bild**, eine bestimmte **Vorstellung**; das sollte sich eine **gute Übersetzung** zunutze machen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Als Denkanstoß vergleiche und bewerte

Deutsch	Englisch	Verunglückt
Betriebssystem	Operating System	Operationssystem
Schnittstelle	Interface	Zwischengesicht
s. Übung 13.7.3(1)	Social Media	Soziale Medien
Taste	Key	Schlüssel
Keller	Stack	Stapel
Bildschirm	Screen	
Feldtyp, -variable	Array type, variable	
Aktualisierung	Update	updaten, upgedatet
Anwender, Nutzer	User	User, Userin
Am/vom Netz	Online/offline	online/offline
Informatik	Computer Science	Comp.wissenschaft(en)
Körper	Field	Feld, Acker
Steuerung	Controlling	Kontrolle, kontrollieren
Nachrichtendienst	Intelligence Service	Intelligenzdienst
Urknall	Big Bang	Großer Knall
Einarmiger Bandit	Slot Machine	Schlitzmaschine

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17

# Bild- und wirkmächtige Übersetzungen

...erfordern Kreativität und Sprachgefühl!

...nur Mut dazu!

Der Unterschied zwischen dem richtigen Wort  
und dem beinahe richtigen Wort ist derselbe wie  
zwischen dem Blitz und dem Glühwürmchen.

Mark Twain (1835-1910)  
amerikan. Schriftsteller

Siehe auch:

- ▶ [Anglizismen-Index, Ausgabe 2019](#). IFB Verlag Deutsche Sprache GmbH, Paderborn, 2019. <http://www.ifb-verlag.de/>  
...mit deutschen Entsprechungen für etwa 7.500 Anglizismen.
- ▶ [Anglizismenübersetzung online](#)  
<https://vds-ev.de/denglisch-und-anglizismen/anglizismenindex/ag-anglizismenindex/>

# Übungsaufgabe 13.7.1

...Hardware, Software.

1. Warum vermutlich ist im Englischen das Begriffspaar  
Hardware/Software  
geprägt worden?
2. Was könnten passende deutsche Begriffspaare sein, die diesem Ursprung folgend nachgebildet sind?
3. Was möglicherweise könnten bildmächtigere deutsche Begriffspaare sein, die losgelöst vom englischen Sprachbild von einer für die deutsche Begriffsbildung passenderen bildlichen Vorstellung, Metapher abgeleitet sind? Im Sinne von Mark Twain [Blitze](#), nicht nur [Glühwürmchen](#)? (s.a. Alvaro Videla. *Metaphors We Compute By*. Communications of the ACM 60(10):42-45, 2017)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe 13.7.2

...Computer, Laptop, Notebook, Workstation.

- ▶ Worauf geht im Englischen die Bezeichnung **Computer** zurück? Was ist entsprechend ein passender dt. Begriff?
- ▶ Für **Laptop** wird gelegentlich **Klapprechner** vorgeschlagen. Gefällt Ihnen der Vorschlag? Wenn ja, warum? Wenn nein, warum nicht?
- ▶ Gelten Ihre Gründe für oder wider **Klapprechner** auch für oder wider **Klapptisch**, **Klappbett**, **Klappsitz**, **Klappliege**, **Klappsessel**, **Klappstuhl**, **Klapptür**, **Klapproller**, **Klapprad**, **Klappbrücke**, **Klappverdeck**, **Klappmesser** u.ä.?
- ▶ Ist **Notizbuch** ein angemessener Begriff für **Notebook**? Transportiert **Notizbuch** ein 'richtiges' Bild, eine 'richtige' Vorstellung vom gemeinten Gegenstand? Gilt dies für **Notebook** im Englischen?
- ▶ Was ist mit **Workstation**? Was transportiert dieser Begriff? Was wäre eine passende dt. Entsprechung?

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe 13.7.3

...Social media, social event, artificial intelligence — soziale Medien, soziale Veranstaltung, artifizielle Intelligenz.

## 1. Vergleiche:

- ▶ Sozialhilfe, Sozialwohnung, Sozialgesetzgebung, Sozialministerium, Sozialladen, Sozialarbeit, Sozialarbeiter, soziale Sicherheit, soziale Verantwortung, soziales Gewissen, soziale Medien, soziale Veranstaltung,...

Welche Begriffe fallen aus der Reihe? Warum? Was könnten treffendere Begriffe dafür sein?

## 2. Vergleiche:

- ▶ Künstliche Intelligenz, artifizielle Intelligenz.

Könnte **artifizielle Intelligenz** sogar als ge Glücktere Entsprechung von **artificial intelligence** gelten als **künstliche Intelligenz**? Wenn ja, warum? In welchem Sinn?

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17



# Plan D, für mehr Prägnanz in der Wissenschaft



## Für mehr **Prägnanz** in der Wissenschaft.

*English:* **science**

*Latina:* **scientia**

*Français:* **science**

*Español:* **ciencia**

*Italiano:* **scienza**

*Deutsch:* **Wissenschaft**

Deutsch macht einen  
entscheidenden **Unterschied**.

Das Bild von Rüdiger Knappe, Zitat: Bild von Rüdiger Knappe, Zitat: Bild von Rüdiger Knappe, Zitat: Bild von Rüdiger Knappe.

Deutsch schafft Wissen.



DAAD

Deutscher Akademischer Austausch Dienst  
German Academic Exchange Service

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 13.8

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 13 (1)

-  Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised Edn., North Holland, 1984. (Kapitel 13, Reduction Strategies)
-  Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2. Auflage, 1998. (Kapitel 7.1, Lazy Evaluation)
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 7.1, Lazy evaluation)
-  Richard Bird, Phil Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Kapitel 6.2, Models of Reduction; Kapitel 6.3, Reduction Order and Space)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 13 (2)

-  Gilles Dowek, Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Springer-V., 2011. (Kapitel 2.3, Reduction Strategies)
-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 2.1, Parameterübergabe und Auswertungsstrategien)
-  Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004. (Kapitel 3, Reduction; Kapitel 8.1, Reduction Machines)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 15, Lazy evaluation; Kapitel 15.2, Evaluation strategies; Kapitel 15.7, Strict application)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10




Kap. 11

Teil V



Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 13 (3)

-  Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011. (Kapitel 4.4, Applicative Order Reduction; Kapitel 8, Evaluation; Kapitel 8.2, Normal Order; Kapitel 8.3, Applicative Order; Kapitel 8.8, Lazy Evaluation)
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 3.1, Reduction Order)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 17.1, Lazy evaluation; Kapitel 17.2, Calculation rules and lazy evaluation)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 13 (4)

-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.  
(Kapitel 17.1, Lazy evaluation; Kapitel 17.2, Calculation rules and lazy evaluation)
-  Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008. (Kapitel 7, Naming; Kapitel 7.1, Parameter Passing)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10




Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 13 (5)

-  Allen B. Tucker (Editor-in-Chief). *Computer Science Handbook*. Chapman & Hall/CRC, 2004. (Kapitel 92, Functional Programming Languages (History of Functional Languages, Pure vs. Impure Functional Languages, Non-strict Functional Languages, Scheme, Standard ML, and Haskell, Research Issues in Functional Programming, etc.))
-  Alvaro Videla. *Metaphors We Compute By*. Communications of the ACM 60(10):42-45, 2017.
-  Reinhard Wilhelm, Helmut Seidl. *Compiler Design – Virtual Machines*. Springer-V., 2010. (Kapitel 3.2, A Simple Functional Programming Language – Evaluation Strategies)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kap. 13.7 (6)



*Anglizismen-Index, Ausgabe 2018.* IFB Verlag Deutsche Sprache GmbH, Paderborn, 2018. Suchfunktion im Netz zugänglich unter: <https://vds-ev.de/denglisch-und-anglizismen/anglizismenindex>



Verein Deutsche Sprache – Deutsch in der Wissenschaft.  
<https://vds-ev.de>



ADAWiS – Arbeitskreis Deutsch als Wissenschaftssprache e.V.  
<http://adawis.de/start>



Deutsch in den Wissenschaften – Deutsch als Wissenschaftssprache. Gemeinsame Erklärung der Präsidenten der Alexander von Humboldt-Stiftung (AvH), des Deutschen Akademischen Austauschdienstes (DAAD), des Goethe-Instituts (GI) und der Hochschulrektorenkonferenz (HRK), 18.02.2009.  
<https://www.goethe.de/lhr/prj/diw/dos/de7753902.htm>



# Kapitel 14

## Typprüfung, Typinferenz

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 14.1

## Motivation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Typisierte Programmiersprachen

...teilen sich in Sprachen mit

- ▶ schwacher (Typprüfung zur Laufzeit)
- ▶ starker (Typprüfung, -inferenz zur Übersetzungszeit)

Typisierung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

...stark typisierter Programmiersprachen:

- ▶ **Verlässlicherer Code:** Der Nachweis der **Typkorrektheit** eines Programms ist ein **Korrektheitsbeweis** für ein Programm auf dem **Abstraktionsniveau von Typen**. Viele Programmierfehler können dadurch schon zur Übersetzungszeit entdeckt werden.
- ▶ **Effizienterer Code:** Keine Typprüfungen zur Laufzeit nötig.
- ▶ **Effektivere Programmentwicklung:** Typinformation ist **Programmdokumentation** und vereinfacht **Verstehen**, **Wartung** und **Weiterentwicklung** eines Programms, z.B. auch bei der Suche nach vordefinierten Bibliotheksfunktionen: "**Gibt es eine Funktion, die alle Duplikate aus einer Liste entfernt?**" erlaubt (in Haskell), die Suche einzuschränken auf Funktionen mit Typ  $((Eq\ a) \Rightarrow [a] \rightarrow [a])$ .

# Haskell

...ist eine **stark typisierte** Programmiersprache.

Dabei gilt:

- ▶ Jeder **gültige** Ausdruck hat einen **definierten Typ**; gültige Ausdrücke heißen **wohlgetypt**.
- ▶ **Typen** gültiger Ausdrücke können sein:
  - ▶ **Monomorph**  
`fac :: Integer -> Integer`
  - ▶ **Parametrisch polymorph** (**uneingeschränkt polymorph**)  
`flip :: (a -> b -> c) -> (b -> a -> c)`
  - ▶ **Ad hoc polymorph** (**eingeschränkt polymorph**)  
`elem :: Eq a => a -> [a] -> Bool`
- ▶ **Typen** können angegeben sein:
  - ▶ **explizit**: **Typprüfung** (grundsätzlich) ausreichend.
  - ▶ **implizit**: **Typinferenz** erforderlich.

# Typprüfung, Typinferenz

...sind **Schlüsselfertigkeiten** von Übersetzern, Interpretierern.

Betrachte den Ausdruck:

```
magicType = let
    pair x y z = z x y
    f y = pair y y
    g y = f (f y)
    h y = g (g y)
in h (\x->x)
```

...zur Veranschaulichung der **Mächtigkeit** automatischer Typinferenzverfahren.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Veranschaulichung (1)

...automatische Typinferenz in Hugs mit dem Kommando `:t` liefert:

```
Main>:t magicType
```

```
magicType ::
```

```
(((((a -> a) -> (a -> a) -> b) -> b) ->
(((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) ->
(((a -> a) -> (a -> a) -> b) -> b) ->
(((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) -> d) -> d) ->
((((a -> a) -> (a -> a) -> b) -> b) -> ((a -> a) ->
(a -> a) -> b) -> b) -> c) -> c) -> (((a -> a) ->
(a -> a) -> b) -> b) -> ((a -> a) ->
(a -> a) -> b) -> b) -> c) -> c) -> d) -> d) -> e) -> e
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Veranschaulichung (2)

...Klammerebenen farblich hervorgehoben:

```
Main>:t magicType
```

```
magicType ::
```

```
(((((a -> a) -> (a -> a) -> b) -> b) ->
(((a -> a) -> (a -> a) -> b) -> b) -> c) ->
(((a -> a) -> (a -> a) -> b) -> b) ->
(((a -> a) -> (a -> a) -> b) -> b) -> c) -> d) -> d) ->
(((a -> a) -> (a -> a) -> b) -> b) -> ((a -> a) ->
(a -> a) -> b) -> b) -> c) -> c) -> (((a -> a) ->
(a -> a) -> b) -> b) -> ((a -> a) ->
(a -> a) -> b) -> b) -> c) -> c) -> d) -> d) -> e) -> e
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Veranschaulichung (3)

...Klammerebenen farblich durchgezählt lassen grobe 'Strukturen' erkennen:

```
Main>:t magicType
```

```
magicType ::
```

```
(1
  (1(1(1(1(1(a -> a) -> (a -> a) -> b1) -> b 1) ->
    (2(2(a -> a) -> (a -> a) -> b2) -> b2) -> c1) -> c1) ->
    (2(2(3(3(a -> a) -> (a -> a) -> b3) -> b3) ->
    (4(4(a -> a) -> (a -> a) -> b4) -> b4) -> c2) -> c2) ->
    d1) -> d
  1)
-> (2(2(3(3(5(5(a -> a) -> (a -> a) -> b5) -> b5) ->
    (6(6(a -> a) -> (a -> a) -> b6) -> b6) -> c3) -> c3) ->
    (4(4(7(7(a -> a) ->
      (a -> a) -> b7) -> b7) -> (8(8(a -> a) ->
      (a -> a) -> b8) -> b8) -> c4) -> c4) -> d2) -> d
    2) -> e1) -> e
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Veranschaulichung (4)

...wobei es auch bleibt.

```
(1
  (1
    (1
      (1
        (1(a → a) → (a → a) → b
          1) → b
        1) → (2
          (2(a → a) → (a → a) → b
            2) → b
          2) → c
        1) → c
      1) → (2
        (2
          (3
            (3(a → a) → (a → a) → b
              3) → b
            3) → (4
              (4(a → a) → (a → a) → b
                4) → b
              4) → c
            2) → c
          2) → d
        1) → d
      1)
    → (2(2(3(3(5(5(a → a) → (a → a) → b5) → b5) →
      (6(6(a → a) → (a → a) → b6) → b6) → c3) → c3) →
      (4(4(7(7(a → a) → (a → a) → b7) → b7) →
      (8(8(a → a) → (a → a) → b8) → b8) → c4) → c4) → d2) → d2) → e
    1) → e
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Automatische Typprüfung, Typinferenz

...der Typ von `magicType` ist fraglos komplex.

Wie gelingt es Übersetzern, Interpretierern, Typen von Ausdrücken wie `magicType` **automatisch** zu inferieren?

**Informell:** Durch Auswertung von

- ▶ **Kontextinformationen** in Ausdrücken, Funktionsdefinitionen und Typklassen.

**Methoden** und **Werkzeuge** dafür:

- ▶ Typanalyse, Typprüfung
- ▶ Typsysteme, Typinferenz
- ▶ Unifikation

...die wir als nächstes (beispielgetrieben) näher betrachten.

# Kapitel 14.2

## Monomorphe Typprüfung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Monomorphe Typprüfung

...liefert als Ergebnis: Ein gegebener **Ausdruck** ist

- ▶ **wohlgetypt**, d.h. hat einen eindeutig bestimmten konkreten Typ.
- ▶ **nicht wohlgetypt**, d.h. hat überhaupt keinen Typ.

# Vereinbarung

...für die folgenden Beispiele.

Polymorphie parametrisch oder überladen polymorpher vordefinierter Funktionen in Haskell wird durch geeignete Typindizierung syntaktisch aufgelöst wie nachstehend angedeutet:

►  $+_{Int}$ ,  $+_{Integer}$ ,  $*_{Double}$ ,  $length_{Char}$ ,...

sind wie folgt typisiert gedacht:

►  $+_{Int} :: Int \rightarrow Int \rightarrow Int$

►  $+_{Integer} :: Integer \rightarrow Integer \rightarrow Integer$

►  $*_{Double} :: Double \rightarrow Double \rightarrow Double$

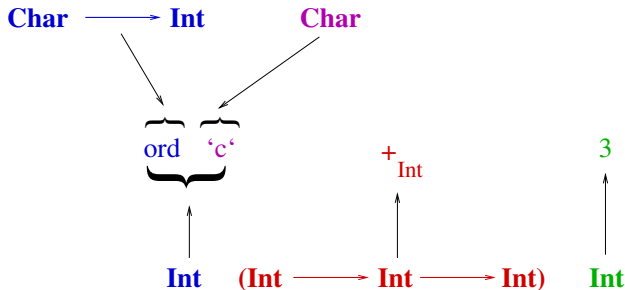
►  $length_{Char} :: [Char] \rightarrow Int$

► ...

# Typprüfung für Ausdrücke (1)

Beispiel 1: Betrachte den Ausdruck  $(\text{ord } 'c' +_{\text{Int}} 3)$ .

Die **Auswertung** des **Ausdruckskontexts** erlaubt

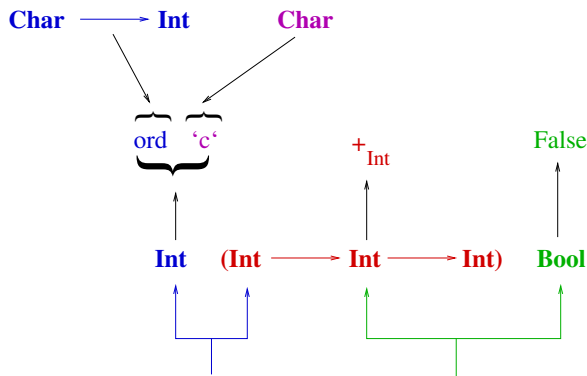


...Typprüfung **korrekte** Typung nachzuweisen!

# Typprüfung für Ausdrücke (2)

Beispiel 2: Betrachte den Ausdruck  $(\text{ord } 'c' +_{\text{Int}} \text{False})$ .

Die Auswertung des Ausdruckskontexts erlaubt



Erwarteter und tatsächlicher Typ, **Int**,  
stimmen überein: **Typkorrekt!**

Erwarteter Typ, **Int**, und tatsächlicher  
Typ, **Bool**, stimmen nicht überein: **Typinkorrekt!**

...Typprüfung **inkorrekte** Typung aufzudecken!



# Typprüfung monomorpher Fkt.-Definitionen

...sei  $f$  **monomorphe** Funktionsdefinition:

```
f :: t1 -> t2 -> ... -> tk -> t
f m1 m2 ... mk
| w1 = a1
| w2 = a2
...
| wn = an
```

...für die **Kontextauswertung** zur **Typprüfung** für  $f$  sind 3 **Eigenschaften** heranzuziehen:

1. Jeder **Wächter**  $w_i$  muss vom Typ **Bool** sein.
2. Jeder **Ausdruck**  $a_i$  muss vom Typ  $t$  sein.
3. Das **Muster** jedes Parameters  $m_i$  muss **konsistent** mit dem zugehörigen Typ  $t_i$  sein.

# Musterkonsistenz, Musterpassung

## Informell:

Ein Muster  $\mu$  ist **konsistent** mit einem Typ  $\tau$ , wenn die auf  $\mu$  passenden Werte vom Typ  $\tau$  sind.

## Detaillierter (vgl. Kap. 6):

- ▶ Eine **Variable** ist mit jedem Typ konsistent.
- ▶ Ein **Literal** oder **Konstante** ist mit ihrem Typ konsistent.
- ▶ Ein Listenmuster  $(p:q)$  ist konsistent mit dem Typ  $[t]$ , wenn  $p$  mit dem Typ  $t$  und  $q$  mit dem Typ  $[t]$  konsistent ist.
- ▶ ...

## Beispiele:

- ▶ Das Muster  $(42:xs)$  ist konsistent mit dem Typ  $[Int]$ .
- ▶ Das Muster  $(x:xs)$  ist konsistent mit jedem **Listentyp**.

# Kapitel 14.3

## Polymorphe Typprüfung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Polymorphe Typprüfung

...liefert als Ergebnis: Ein gegebener **Ausdruck** ist

- ▶ **wohlgetypt** und steht (abkürzend) für **einen** oder **mehrere**, möglicherweise **unendlich viele** konkrete Typen.
- ▶ **nicht wohlgetypt**, d.h. hat überhaupt keinen Typ.

Schlüssel zur **algorithmischen** Lösung ist das

- ▶ **Lösen** von **Kontextinformationstypbedingungssystemen** (engl. **constraint satisfaction**)

auf Grundlage der **Unifikation** von **Typausdrücken**.

# Polymorphe Typprüfung (1)

**Beispiel 1:** Betrachte die Funktionssignatur von `length` mit dem **polymorphen** Typ `([a] -> Int)`:

```
length :: [a] -> Int
```

...informell steht `([a] -> Int)` abkürzend für die unendliche Menge konkreter Typen `([ $\tau$ ] -> Int)`, wobei  $\tau$  Platzhalter für einen beliebigen **monomorphen** Typ ist:

```
([Int] -> Int)
([Bool,Char]) -> Int
([(String -> String)] -> Int)
([(Bool -> Bool -> Bool)] -> Int)
...
```

# Polymorphe Typprüfung (2)

In **Aufrufkontexten** wie:

```
length [length [1,2,3], length [True,False,True],  
        length [], length [(+),(*),(-)]]  
length [(True,'a'), (False,'q'), (True,'o')]  
length [reverse, ("Felix" ++), tail, init]  
length [(&&), (||), xor, nand, nor]
```

...kann der konkrete **monomorphe** Typ von Anwendungen von **length** erschlossen werden:

```
length :: [Int] -> Int  
length :: [(Bool,Char)] -> Int  
length :: [(String -> String)] -> Int  
length :: [(Bool -> Bool -> Bool)] -> Int
```

**Ausnahme:** Der Aufruf (**length []**) erlaubt nur auf **length :: [a] -> Int** zu schließen. **Übungsaufgabe:** Warum?

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

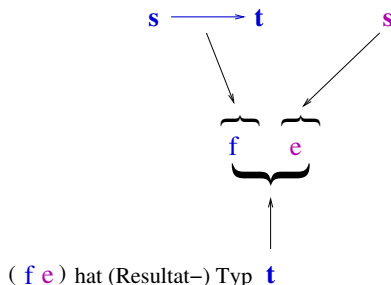
Teil V

Kap. 12

Kap. 13/17

# Polymorphe Typprüfung (3)

Beispiel 2: Betrachte den applikativen Ausdruck  $(f\ e)$ :



Ist weitere Kontextinformation für  $f$  und  $e$  nicht vorhanden, liefert die **Auswertung** des **Anwendungskontexts** von  $(f\ e)$  die **allgemeinst möglichen** Typen von  $e$ ,  $f$  und  $(f\ e)$  wie folgt:

- ▶  $e \quad :: \quad s$
- ▶  $f \quad :: \quad s \rightarrow t$
- ▶  $(f\ e) \quad :: \quad t$

# Polymorphe Typprüfung (4)

Beispiel 3: Betrachte die Funktionsgleichung:

$$f(x, y) = (x, ['a' .. y])$$

Die **Auswertung** des **Anwendungskontexts** ergibt: Funktion **f** erwartet als Argument **Paare**, an deren

- ▶ **1-te Komponente** keine Bedingung gestellt ist, die also von einem beliebigen Typ sein darf.
- ▶ **2-te Komponente** eine Bedingung gestellt ist: **y** muss vom Typ **Char** sein, da **y** als Schranke des Zeichenreihenwerts **['a' .. y]** benutzt wird.

Beides zusammen erlaubt den **allgemeinsten Typ** von **f** zu erschließen:

$$f :: (a, \text{Char}) \rightarrow (a, [\text{Char}])$$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Polymorphe Typprüfung (5)

Beispiel 4: Betrachte die Funktionsgleichung:

$$g(m, zs) = m + \text{length } zs$$

Die **Auswertung** des **Anwendungskontexts** ergibt: Funktion  $g$  erwartet als Argument **Paare**, an deren Komponenten folgende Bedingungen gestellt sind:

- ▶ **1-te Komponente:**  $m$  muss von einem numerischen Typ sein, da  $m$  als Operand von  $(+)$  verwendet wird.
- ▶ **2-te Komponente:**  $zs$  muss vom Typ  $[b]$  sein, da  $zs$  als Argument der Funktion  $\text{length}$  verwendet wird, die den Typ  $([b] \rightarrow \text{Int})$  hat.

Beides zusammen erlaubt den **allgemeinsten Typ** von  $g$  zu erschließen:

$$g :: (\text{Int}, [b]) \rightarrow \text{Int}$$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Polymorphe Typprüfung (6)

Die Beispiele zeigen, dass wie im **monomorphen** Fall die **Anwendungskontexte** von Ausdrücken und Funktionsdefinitionen implizit ein

- ▶ **System** von **Typbedingungen** festlegen.

Das **Typprüfungsproblem** reduziert sich so auf die Bestimmung der

- ▶ **allgemeinst möglichen** Typausdrücke, so dass **keine** Bedingung verletzt ist.

# Polymorphe Typprüfung (7)

Beispiel 5: Betrachte die Komposition  $(g \circ f)$  mit  $f, g$  aus Bsp. 3 und 4.

In Funktionskompositionen  $(h' \circ h)$  ist

- das **Resultat** der Anwendung von  $h$  das **Argument** der Anwendung von  $h'$ .

Die **Auswertung** des **Anwendungskontexts** von  $(g \circ f)$  gegeben durch die Gleichungen in Bsp. 3 und 4 ergibt zusätzlich:

- Das Resultat von  $f$  ist vom Typ  $(a, [\text{Char}])$ .
- Das Argument von  $g$  ist vom Typ  $(\text{Int}, [b])$ .

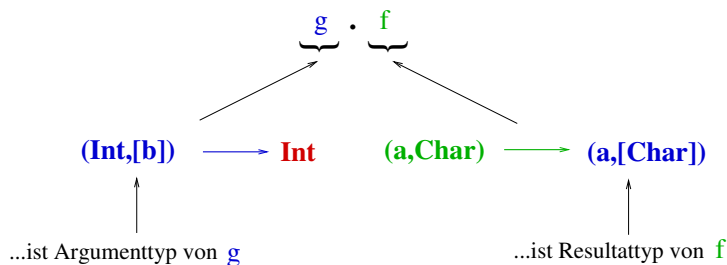
Damit verbleiben noch zu bestimmen: Die

- **allgemeinst möglichen Typen** für die Typvariablen  $a$  und  $b$ , die obige 3 Bedingungen erfüllen.

Der Schlüssel hierfür: **Unifikation**.

# Polymorphe Typprüfung (8)

Veranschaulichung des Unifikationsvorgehens:

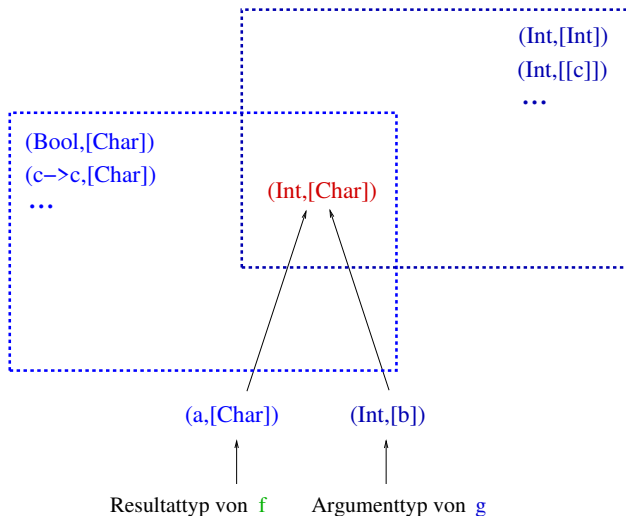


...Unifikation löst die 3 Bedingungen in Kombination auf und liefert  $Int$  als allgemeinst möglichen Typ für  $a$ ,  $Char$  für  $b$  und somit  $((Int, Char) \rightarrow Int)$  für  $(g \cdot f)$ , d.h.:

$(g \cdot f) :: (Int, Char) \rightarrow Int$

# Polymorphe Typprüfung (9)

Veranschaulichung des **Unifikations**vorgehens:



# Instanz, gem. Instanz, Unifikat, Unifikator (1)

Ein **Typausdruck**  $a$  ist (**Typ-**)

- ▶ **Instanz** eines Typausdrucks  $a'$ , wenn  $a$  aus  $a'$  durch konsistentes Ersetzen (oder Substitution) von Typvariablen mit Typausdrücken entsteht.
- ▶ **gemeinsame Instanz** einer Menge  $M$  von Typausdrücken, wenn  $a$  Instanz von allen Typausdrücken aus  $M$  ist.
- ▶ **allgemeinste gemeinsame Instanz** einer Menge  $M$  von Typausdrücken, wenn  $a$  gemeinsame Instanz von  $M$  ist und für alle anderen gemeinsamen Instanzen  $b$  von  $M$  gilt, dass  $b$  Instanz von  $a$  ist;  $a$  heißt dann (allgemeinstes) **Unifikat** von  $M$ , die zugehörige Substitution (allgemeinster) **Unifikator** von  $M$ .

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Instanz, gem. Instanz, Unifikat, Unifikator (2)

...gleichwertig: Ein Typausdruck  $a$  ist

- ▶ **Instanz** eines Typausdrucks  $a'$ , wenn  $a'$  sich zu  $a$  spezialisieren lässt; wenn  $a$  eine Teilmenge von Typen von  $a'$  beschreibt.
- ▶ **gemeinsame Instanz** einer Menge  $M$  von Typausdrücken, wenn jeder Typausdruck  $a'$  aus  $M$  sich zu  $a$  spezialisieren lässt; wenn  $a$  eine Teilmenge von Typen jedes Typausdrucks aus  $M$  beschreibt; wenn  $a$  eine Teilmenge des Durchschnitts der von den Typausdrücken aus  $M$  beschriebenen Typmengen beschreibt.
- ▶ **allgemeinste gemeinsame Instanz** einer Menge  $M$  von Typausdrücken, wenn  $a$  gemeinsame Instanz von  $M$  ist und für alle anderen gemeinsamen Instanzen  $b$  von  $M$  gilt, dass sich  $a$  zu  $b$  spezialisieren lässt; dass jede andere gemeinsame Instanz  $b$  von  $M$  eine Teilmenge der Typen von  $a$  beschreibt.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Gemeinsame Instanz v. $M \not\Rightarrow$ Unifikat v. $M$ (1)

Betrachte folgendes Beispiel:

Die Typausdrücke

►  $([Bool], [[Bool]]), ([[e]], [[e]]), ([d], [[d]])$

sind gemeinsame Instanzen (oder Spezialisierungen) der Typausdrücke

►  $(a, [a])$  und  $([b], c)$

unter den Substitutionen

►  $[Bool], [[e]], [d]$  für  $a$

►  $Bool, [e], d$  für  $b$

►  $[[Bool]], [[e]], [[d]]$  für  $c$ .



# Gemeinsame Instanz v. $M \not\Rightarrow$ Unifikat v. $M$ (2)

In Substitutionsschreibweise (vgl. Kap. 12.3.1):

- ▶  $(a, [a])[ \text{Bool}/a ] = ([\text{Bool}], [[\text{Bool}]])$   
 $(a, [a])[ [e]/a ] = ([e], [[e]])$   
 $(a, [a])[ d/a ] = ([d], [[d]])$
- ▶  $(([b], c)[ \text{Bool}/b, [\text{Bool}]/c ] = ([\text{Bool}], [[\text{Bool}]])$   
 $(([b], c)[ e/b, [e]/c ] = ([e], [[e]])$   
 $(([b], c)[ d/b, [d]/c ] = ([d], [[d]])$

# Gemeinsame Instanz v. $M \not\Rightarrow$ Unifikat v. $M$ (3)

Weiters sind beide Typausdrücke

►  $([Bool], [[Bool]]), ([[e]], [[e]])$

Instanzen (oder Spezialisierungen) des Typausdrucks

►  $([d], [[d]])$

unter den Substitutionen  $Bool, [e]$  für  $d$ :

►  $([d], [[d]]) [ Bool/d ] = ([Bool], [[Bool]])$

$([d], [[d]]) [ e/d ] = ([[e]], [[e]])$

Umgekehrt ist der Typausdruck

►  $([d], [[d]])$

keine Instanz (oder Spezialisierung) von einem der Ausdrücke

►  $([Bool], [[Bool]]), ([[e]], [[e]])$ .

# Gemeinsame Instanz v. $M \not\Rightarrow$ Unifikat v. $M$ (4)

Zusammengefasst:

Die Typausdrücke  $([Bool], [[Bool]])$ ,  $([[e]], [[[e]]])$

- ▶ sind **gemeinsame Instanzen** der Typausdrucksmenge  
 $M =_{df} \{(a, [a]), ([b], c), ([d], [[d]])\}$
- ▶ jedoch **keine allgemeinsten Instanzen** von  $M$ , d.h. **keiner** der beiden Ausdrücke ist **Unifikat** von  $M$ .

Der Typausdruck  $([d], [[d]])$  ist

- ▶ die **allgemeinste gemeinsame Instanz** und damit das **Unifikat** von  $M$ .

Insgesamt ist damit gezeigt:

- ▶ Die Eigenschaft '**gemeinsame Instanz**' einer Menge von Typausdrücken **impliziert nicht** die Eigenschaft '**Unifikat**' dieser Menge.

# Unifikation, Unifikationsaufgabe

...ist die Bestimmung der

- ▶ **allgemeinsten gemeinsamen (Typ-) Instanz** (engl. **most general common (type) instance**) einer Menge von Typausdrücken und der **zugehörigen Substitution**.

Informell: **Unifikation**

- ▶ **bestimmt** allgemeinstmögliche mehrere Typbedingungen zugleich erfüllende Typausdrücke, die **allgemeinste gemeinsame Instanz** einer Menge von Typausdrücken sind.
- ▶ wertet dafür **Kontextbedingungen** in **Kombination** aus.
- ▶ führt i.a. zu **polymorphen** Typausdrücken.
- ▶ kann **fehschlagen**.

# Unifikation bestimmt allgemeinste Instanzen

...unter Auswertung von **Kontextbedingungen** in **Kombination**.

Illustriert an **Beispiel 5**: Unifikation bestimmt unter kombinierter Auswertung der **Kontextbedingungen**

$(g \ . \ f)$

$f \ (x,y) = (x, ['a' \ .. \ y])$

$g \ (m,zs) = m + \text{length } zs$

den Typausdruck

►  $(\text{Int}, [\text{Char}])$

als **allgemeinste gemeinsame Instanz** der Menge **M** von Typausdrücken

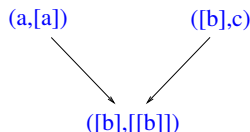
►  $M =_{df} \{ (a, [\text{Char}]), (\text{Int}, [b]) \}$

unter der **Substitution** **Int** für **a** und **Char** für **b**:

►  $(a, [\text{Char}]) [\text{Int}/a] = (\text{Int}, [b]) [\text{Char}/b] = (\text{Int}, [\text{Char}])$

# Unifikation liefert polymorphen Typ

Beispiel:



$([b],[[b]])$ , die allgemeinste gemeinsame Instanz von  $(a,[a])$  und  $([b],c)$ .

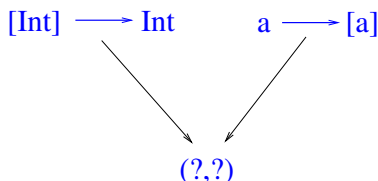
Für die Unifikation von  $(a,[a])$  und  $([b],c)$  verlangt die **Kontextbedingung**

- ▶  $(a,[a])$ : Die 2-te Komponente ist eine Liste von Elementen des Typs der 1-ten Komponente.
- ▶  $([b],c)$ : Die 1-te Komponente ist von einem Listentyp.

Zusammen impliziert das: Die **allgemeinste gemeinsame Instanz** von  $(a,[a])$  und  $([b],c)$  ist der (nichtmonomorphe) polymorphe Typausdruck  $([b],[[b]])$ .

# Unifikation schlägt fehl (d.h. ist nicht möglich)

Beispiel:



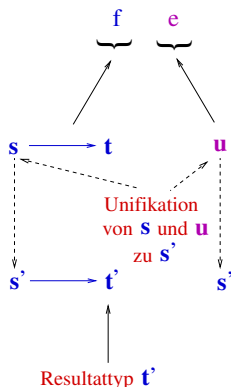
Für die Unifikation von  $([Int] \rightarrow [Int])$  und  $(a \rightarrow [a])$  verlangt die **Unifikation** der

- ▶ **Argumenttypen:**  $a$  ist vom Typ  $[Int]$  ist.
- ▶ **Resultattypen:**  $a$  ist vom Typ  $Int$  ist.

Das schließt sich aus und ist nicht zugleich erfüllbar, eine gemeinsame Typinstanz existiert nicht: **Unifikation** schlägt **fehl**.

# Typüberprüfung von Fkt.-Termen (1)

Betrachte den applikativen Ausdruck  $(f\ e)$ :



Es gilt: Typkorrektheit von  $(f\ e)$  erfordert nicht Gleichheit von  $s$  und  $u$ ; es reicht, wenn sie **unifizierbar** sind: Unifizierter Typ von  $f$  ist  $(s' \rightarrow t')$ , von  $(f\ e)$  somit  $t'$ .



# Typüberprüfung von Fkt.-Termen (2)

Betrachte den applikativen Ausdruck `(map ord)` mit den Kontextbedingungen:

```
map :: (a -> b) -> [a] -> [b]
ord :: Char -> Int
```

Unifikation der Typausdrücke `(a -> b)` und `(Char -> Int)` liefert als allgemeinst mögliche Typen für `(map ord)` und `map`:

```
(map ord) :: [Char] -> [Int]
map :: (Char -> Int) -> [Char] -> [Int]
```

## Typüberprüfung von Fkt.-Termen (3)

Betrachte den applikativen Term `(foldr (+) 0 [3,5,34])` mit den **Kontextbedingungen**:

`(foldr (+) 0 [1,2,3,5,7,11,13]) :: Int (->> 42)`

`foldr f s [] = s`

`foldr f s (x:xs) = f x (foldr f s xs)`

Für die Typen von `(foldr (+) 0 [3,5,34])` und `foldr` liefert das:

`(foldr (+) 0 [1,2,3,5,7,11,13]) :: Int`

`foldr :: (Int -> Int -> Int) -> Int -> [Int] -> Int`

Naiv suggeriert dies für den **'allgemeinsten'** Typ von `foldr`:

`foldr :: (a -> a -> a) -> a -> [a] -> a`

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Typüberprüfung von Fkt.-Termen (4)

...eine genauere **Überlegung** liefert:

$\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

Veranschaulichung:

$$\begin{aligned}\text{foldr } f \text{ } s \text{ } [] &= s \\ \text{foldr } f \text{ } s \text{ } (x:xs) &= f \text{ } x \text{ } (\text{foldr } f \text{ } s \text{ } xs)\end{aligned}$$

# Typprüfung polymorpher Fkt.-Definitionen

...sei  $f$  parametrisch polymorphe Funktionsdefinition:

$$f :: t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_k \rightarrow t$$
$$f \ m_1 \ m_2 \ \dots \ m_k$$
$$| \ w_1 = a_1$$
$$| \ w_2 = a_2$$
$$\dots$$
$$| \ w_n = a_n$$

...für die Kontextauswertung zur Typprüfung für  $f$  sind 3 Eigenschaften heranzuziehen:

1. Jeder Wächter  $w_i$  muss vom Typ `Bool` sein.
2. Jeder Ausdruck  $a_i$  muss von einem Typ  $s_i$  sein, der mindestens (! – umgekehrt im Aufruffall) so allgemein ist wie der Typ  $t$ , d.h.  $t$  muss eine Instanz von  $s_i$  sein.
3. Das Muster jedes Parameters  $m_i$  muss konsistent mit dem zugehörigen Typ  $t_i$  sein.

# Unifikation mit Konstanten, Variablen (1)

...Konstanten und Variablen werden in Haskell bei Unifikation unterschiedlich behandelt.

Betrachte folgendes Beispiel:

Der Ausdruck `a` kann erfolgreich getypt werden, die davon abgeleitete Funktionsabstraktion `f` hingegen nicht:

```
a      = length ([] ++ [True])  
        + length ([] ++ [1,2,3]) :: Int
```

```
f xs = length (xs ++ [True])  
        + length (xs ++ [1,2,3])  ↗ Nicht typbar!
```

# Unifikation mit Konstanten, Variablen (2)

Das Beispiel zeigt:

- ▶ **Konstanten** wie `[]` können unterschiedlich getypt in Ausdrücken verwendet werden: In `a` verlangt
  - ▶ 1-te Verwendung von `[]`: `[] :: [Bool]`
  - ▶ 2-te Verwendung von `[]`: `[] :: [Int]`.

Unifikation analysiert für Konstanten wie `[]` beide Vorkommen getrennt und gelingt.

- ▶ **Variablen** wie `xs` dürfen das nicht: In `f` verlangt
  - ▶ 1-te Verwendung von `xs`: `xs :: [Bool]`
  - ▶ 2-te Verwendung von `xs`: `xs :: [Int]`.

Beides zusammen ist unvereinbar. Die verschiedenen Verwendungen von `xs` werden (anders als bei Konstanten) von Unifikation für Variablen nicht getrennt; Unifikation schlägt deshalb für `f` fehl.

## Übung 14.3.1

Zeige, dass die unterschiedliche Behandlung von Konstanten und Variablen durch Unifikation sinnvoll ist.

Zu welchen Widersprüchen würde die vermeintlich naheliegende Typisierung

```
f :: [a] -> Int
```

führen? Würde die starke Typisierung von Haskell erhalten bleiben, die zusichert, dass Laufzeitfehler aufgrund von Typfehlern ausgeschlossen sind?

Überlege dazu, Listen welcher Argumenttypen `f` verkraften müsste und ob ihre Implementierung durch die definierende Gleichung

```
f xs = length (xs++[True]) + length (xs++[1,2,3])
```

das hergäbe und was daraus für starke Typisierung und die daraus folgenden Zusicherungen folgte.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 14.4

## Polymorphe Typprüfung mit Typklassen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Typprüfung mit Typklassen (1)

Betrachte folgende Funktionsdefinition:

```
member []      y = False
member (x:xs) y = (x == y) || member xs y
```

Aus der Auswertung des Kontexts, hier

- ▶ dem Listenmuster `(x:xs)` für das erste Argument
- ▶ dem Funktionsresultat `False` in der 1-ten Gleichung
- ▶ der Benutzung von `(==)` in der 2-ten Gleichung

können wir für den allgemeinsten Typ von `member` schließen:

```
member :: Eq a => [a] -> a -> Bool
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Typprüfung mit Typklassen (2)

Betrachte zusätzlich zu den definierenden Gleichungen von `member` den Ausdruck `e` mit der **Kontextinformation**:

`e :: Ord b => [[b]]`

Gesucht ist nun der allgemeinste Typ des applikativen Ausdrucks `(member e)`.

Naiv ohne Berücksichtigung der **Typklassenkontexte** von `e` und `member`, lieferte dies für die Typen von `(member e)`, `member` und `e`:

```
e :: [[b]]
member :: [[b]] -> [b] -> Bool
(member e) :: [b] -> Bool
```

# Typprüfung mit Typklassen (3)

Mit Berücksichtigung der kombinierten Typklassenkontexte von `member` und `e`:

`(Eq [b], Ord b)`

erhalten wir jedoch für den Typ von `(member e)` zunächst:

`(member e) :: (Eq [b], Ord b) => [b] -> Bool`

...und schließlich nach einer Typklassenkontextanalyse zur Typklassenkontextvereinfachung einfacher:

`(member e) :: Ord b => [b] -> Bool`

# Typklassenkontextanalyse (1)

...Analyse und Typklassenkontextvereinfachung erfolgt mehrschrittig, im Bsp. vom Kontext `(Eq [b], Ord b)` zum Kontext `(Ord b)`:

1. **Herunterbrechen** von Typklassenkontextbedingungen wie `(Eq [b])` auf Bedingungen an Typvariablen wie `b` durch Analyse der involvierten Typklasseninstanzdeklaration wie `instance Eq a => Eq [a] where...`
2. **Wiederholen** von Schritt 1) bis keine Instanzdeklaration mehr anwendbar ist.
3. **Weiteres Vereinfachen** des Kontexts aus Schritt 2) durch Auswertung der involvierten Typklassendefinitionen wie `class Eq a => Ord a where....`

# Typklassenkontextanalyse (2)

Für unser Beispiel erhalten wir auf diese Weise:

1. Ausgehend vom Kontext  $(Eq\ [b], Ord\ b)$ , liefert die Analyse der Instanzdeklaration  $(instance\ Eq\ a \Rightarrow Eq\ [a]\ where...)$  die Implikation  $Eq\ b$ , wenn  $Eq\ [b]$ . Das erlaubt  $(Eq\ [b], Ord\ b)$  zu  $(Eq\ b, Ord\ b)$  zu vereinfachen.
2. Keine Instanzdeklaration mehr anwendbar; weiter mit 3).
3. Ausgehend von  $(Eq\ b, Ord\ b)$  aus Schritt 2), liefert die Analyse der Typklassendefinition  $(class\ Eq\ a \Rightarrow Ord\ a\ where...)$  die Implikation  $Ord\ b$ , wenn  $Eq\ b$ . Das erlaubt  $(Eq\ b, Ord\ b)$  zu  $(Ord\ b)$  zu vereinfachen; keine weitere Vereinfachung mehr möglich.

Somit erhalten wir insgesamt für den allgemeinsten Typ des applikativen Ausdrucks  $(member\ e)$ :

►  $(member\ e) :: Ord\ b \Rightarrow [b] \rightarrow Bool$

# Zusammenfassung

...der dreistufige Prozess aus:

- ▶ Unifikation
- ▶ Analyse (einschl. Instanz- und Typklassendeklarationen)
- ▶ Simplifikation

ist das allgemeine Muster für **polymorphe Typprüfung** mit **Typklassen** in **Haskell**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 14.5

## Typsysteme, Typinferenz

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Typsysteme, Typinferenz

Informell:

Typsysteme sind

- ▶ logische Systeme, die uns erlauben, Aussagen der Form 'exp ist Ausdruck vom Typ t' zu formalisieren und sie mithilfe von Axiomen und Regeln des Typsystems zu beweisen.

Typinferenz bezeichnet

- ▶ den Prozess, den Typ eines Ausdrucks automatisch mithilfe der Axiome und Regeln des Typsystems abzuleiten.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Typgrammatik (typischer Ausschnitt)

...erzeugt eine **Typsprache**:

$$\begin{array}{ll} \tau ::= & \textit{Int} \mid \textit{Float} \mid \textit{Char} \mid \textit{Bool} & \text{(Einfacher Typ)} \\ & \mid \alpha & \text{(Typvariable)} \\ & \mid \tau \rightarrow \tau & \text{(Funktionstyp)} \end{array}$$
$$\begin{array}{ll} \sigma ::= & \tau & \text{(Typ)} \\ & \mid \forall \alpha. \sigma & \text{(Typbindung)} \end{array}$$

Sprechweisen:  $\tau$  ist ein **Typ**,  $\sigma$  ein **Typschema**.

# Typsystem (typischer Ausschnitt)

...assoziiert mit jedem (typisierbaren) Ausdruck der Sprache einen **Typ** der Typsprache, wobei  $\Gamma$  eine sogenannte **Typannahme** (oder **Typumgebung**) ist:

Axiome:

$$\text{VAR} \quad \frac{}{\Gamma \vdash \text{var} : \Gamma(\text{var})}$$

$$\text{CON} \quad \frac{}{\Gamma \vdash \text{con} : \Gamma(\text{con})}$$

$$\text{COND} \quad \frac{\Gamma \vdash \text{exp} : \text{Bool} \quad \Gamma \vdash \text{exp}_1 : \tau \quad \Gamma \vdash \text{exp}_2 : \tau}{\Gamma \vdash \text{if exp then exp}_1 \text{ else exp}_2 : \tau}$$

Regeln:

$$\text{APP} \quad \frac{\Gamma \vdash \text{exp} : \tau' \rightarrow \tau \quad \Gamma \vdash \text{exp}' : \tau'}{\Gamma \vdash \text{exp exp}' : \tau}$$

$$\text{ABS} \quad \frac{\Gamma[\text{var} \mapsto \tau'] \vdash \text{exp} : \tau}{\Gamma \vdash \lambda x. \text{exp} : \tau' \rightarrow \tau}$$

...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Typumgebung, substituierte Typumgebung

Typumgebungen sind

- ▶ partielle Abbildungen, die Typvariablen auf Typschemata abbilden.

Ist  $\Gamma$  eine Typumgebung, so ist  $\Gamma[\tau_1/var_1, \dots, \tau_n/var_n]$

- ▶ eine substituierte Typumgebung, die jede Typvariable  $var_i$  auf den Typ  $\tau_i$  abbildet; jede andere Typvariable auf ihren Typ in der Typumgebung  $\Gamma$ .

# Unifikationsalgorithmus (schematisch)

$$\mathcal{U}(\alpha, \alpha) = []$$

$$\mathcal{U}(\alpha, \tau) = \begin{cases} [\tau/\alpha] & \text{falls } \alpha \notin \tau \\ \text{Fehl Schlag} & \text{sonst} \end{cases}$$

$$\mathcal{U}(\tau, \alpha) = \mathcal{U}(\alpha, \tau)$$

$$\mathcal{U}(\tau_1 \rightarrow \tau_2, \tau_3 \rightarrow \tau_4) = \mathcal{U}(U_{\tau_2}, U_{\tau_4})U \text{ mit } U = \mathcal{U}(\tau_1, \tau_3)$$

$$\mathcal{U}(\tau, \tau') = \begin{cases} [] & \text{falls } \tau = \tau' \\ \text{Fehl Schlag} & \text{sonst} \end{cases}$$

## Anmerkung:

- ▶ Die Anwendung der Gleichungen erfolgt sequentiell von oben nach unten.
- ▶  $U$  für (allgemeinster) Unifikator (i.w. eine Substitution).

# Unifikator, allgemeinsten Unifikator

Beispiel: Betrachte die Typausdrücke  $(a \rightarrow (\text{Bool}, c))$  und  $(\text{Int} \rightarrow b)$ .

Durch **scharfes Hinsehen** erkennt man:

Die Substitution  $[\text{Int}/a, \text{Float}/c, (\text{Bool}, \text{Float})/b]$

- ist **ein** Unifikator von  $(a \rightarrow (\text{Bool}, c))$ ,  $(\text{Int} \rightarrow b)$ .

Die Substitution  $[\text{Int}/a, (\text{Bool}, c)/b]$

- ist **der** (allgemeinste) Unifikator von  $(a \rightarrow (\text{Bool}, c))$ ,  $(\text{Int} \rightarrow b)$ .

# Anwendung des Unifikationsalgorithmus

...am Beispiel der Unifikation der Typausdrücke  $(a \rightarrow c)$  und  $(b \rightarrow \text{Int} \rightarrow a)$ .

Rechnen liefert:

$$\begin{aligned} & \mathcal{U}(a \rightarrow c, b \rightarrow \text{Int} \rightarrow a) \\ (\text{mit } U = \mathcal{U}(a, b) = [b/a]) &= \mathcal{U}(Uc, U(\text{Int} \rightarrow a))U \\ &= \mathcal{U}(c, \text{Int} \rightarrow b)[b/a] \\ &= [\text{Int} \rightarrow b/c][b/a] \\ &= [\text{Int} \rightarrow b/c, b/a] \end{aligned}$$

Damit ist der **allgemeinste Unifikator** der beiden Typausdrücke die **Substitution**  $[(\text{Int} \rightarrow b)/c, b/a]$  und das (allgemeinste) **Unifikat** der Typausdrücke  $(a \rightarrow c)$  und  $(b \rightarrow \text{Int} \rightarrow a)$  der Typausdruck:

$$\begin{aligned} \blacktriangleright (b \rightarrow \text{Int} \rightarrow b) &= \\ (a \rightarrow c)[(\text{Int} \rightarrow b)/c, b/a] &= \\ (b \rightarrow \text{Int} \rightarrow a)[(\text{Int} \rightarrow b)/c, b/a] & \end{aligned}$$

# Entscheidend für den Typinferenzalgorithmus

...die **syntaxgerichtete** Anwendung der Regeln des Typinferenzsystems, d.h. es ist

- ▶ stets nur **ein** Axiom oder **eine** Regel anwendbar.

**Schlüssel** dazu: Anpassung des Typinferenzsystems.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Zusammenfassung (1)

Unifikation ist

- ▶ zentral für **polymorphe Typinferenz**.

Das Beispiel der Funktion **magicType** illustriert die

- ▶ **Mächtigkeit automatischer Typinferenz**.

Das wirft die Frage auf:

- ▶ **Lohnt es** (sich die Mühe anzutun), **Typen zu spezifizieren**, wenn (auch derart) komplexe Typen wie im Fall von **magicType** automatisch hergeleitet werden können?

Antwort: **Ja**, denn **Typspezifikationen**

- ▶ sind eine **sinnvolle Kommentierung** des Programms.
- ▶ ermöglichen Interpretierern und Übersetzern **aussagekräftigere Fehlermeldungen** zu erzeugen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17



# Zusammenfassung (2)

Haskell ist stark typisiert:

- ▶ Wohltypisierung von Programmen ist deshalb zur Übersetzungszeit entscheidbar. Fehler zur Laufzeit aufgrund von Typfehlern sind deshalb ausgeschlossen.
- ▶ Typen können, müssen aber vom Programmierer nicht angegeben werden. Übersetzer und Interpretierer inferieren die Typen von Ausdrücken und Funktionsdefinitionen (in jedem Fall) automatisch.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Zusammenfassung (3)

...Leseempfehlungen zu [Typprüfung](#), [Typinferenz](#).

[Funktionale Sprachen](#) allgemein:

- ▶ Anthony J. Field, Peter G. Robinson. [Functional Programming](#). Addison-Wesley, 1988. (Kapitel 7, Type inference systems and type checking)

[Haskell-spezifisch](#):

- ▶ Simon Peyton Jones, John Hughes. [Report on the Programming Language Haskell 98](#).  
<http://www.haskell.org/report/>

# Zusammenfassung (4)

## Überblick Typsysteme:

- ▶ John C. Mitchell. [Type Systems for Programming Languages](#). In Jan van Leeuwen (Hrsg.). *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*. Elsevier Science Publishers, 367-458, 1990.

## Grundlagen polymorpher Typsysteme:

- ▶ Robin Milner. [A Theory of Type Polymorphism in Programming](#). *Journal of Computer and System Sciences* 17:248-375, 1978.
- ▶ Luís Damas, Robin Milner. [Principal Type Schemes for Functional Programming Languages](#). In Conference Record of the 9th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82), 207-218, 1982.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Zusammenfassung (5)

## Unifikation:

- ▶ J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM 12(1):23-42, 1965.

## Typsysteme, Typinferenz:

- ▶ Luca Cardelli. *Basic Polymorphic Type Checking*. Science of Computer Programming 8:147-172, 1987.

# Kapitel 14.6

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10




Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 14 (1)

-  Luca Cardelli. *Basic Polymorphic Type Checking*. Science of Computer Programming 8:147-172, 1987.
-  Luís Damas, Robin Milner. *Principal Type Schemes for Functional Programming Languages*. In Conference Record of the 9th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82), 207-218, 1982.
-  Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 4.7, Type Inference)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10




Kap. 11

Teil V





Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 14 (2)

-  Gilles Dowek, Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Springer-V., 2011. (Kapitel 6, Type Inference; Kapitel 6.1, Inferring Monomorphic Types; Kapitel 6.2, Polymorphism)
-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 5, Typisierung und Typinferenz)
-  Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988. (Kapitel 7, Type inference systems and type checking)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 14 (3)

-  Robin Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences 17:248-375, 1978.
-  John C. Mitchell. *Type Systems for Programming Languages*. In *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*, Jan van Leeuwen (Hrsg.). Elsevier Science Publishers, 367-458, 1990.
-  Simon Peyton Jones (Hrsg.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. [www.haskell.org/definitions](http://www.haskell.org/definitions).
-  J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM 12(1):23-42, 1965.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11




Teil V

Kap. 12

Kap. 13



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 14 (4)

-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 5, Writing a Library: Working with JSON Data – Type Inference is a Double-Edged Sword)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 13, Checking types)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 13, Overloading, type classes and type checking)

# Teil VI

## Weiterführende Konzepte

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 15

## Ein- und Ausgabe

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 15.1

## Motivation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

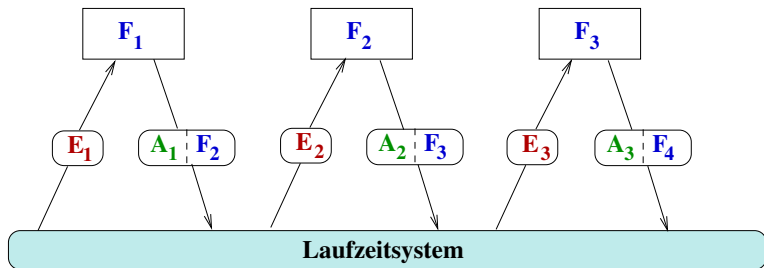
Teil V

Kap. 12

Kap. 13

# Erwartung

...Programme sind **dialog-** und **interaktionsorientiert** dank **Ein-** und **Ausgabemöglichkeiten**:



Peter Pepper. *Funktionale Programmierung*.  
Springer-Verlag, 2003, S. 253.

# Aber

...unsere Programme sind bislang **stapelverarbeitungsorientiert**:



Peter Pepper. *Funktionale Programmierung*.  
Springer-Verlag, 2003, S. 245.

**Dialog** oder **Interaktion** zwischen Benutzer und Programm finden nicht statt:

- ▶ **Eingabedaten** müssen **zu Programmbeginn vollständig** zur Verfügung gestellt werden.
- ▶ **Einmal gestartet**, besteht **keine Möglichkeit** mehr, mit weiteren Eingaben auf das Verhalten oder Ergebnisse des Programms **zu reagieren** und es **zu beeinflussen**.

# Kapitel 15.1.1

## Die Herausforderung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Die Herausforderung

...einen scheinbar unauflöslchen **Widerspruch auflösen** und zu **vereinen**: Den **Umgang**

- ▶ mit **Seiteneffekten** in einer **seiteneffektlosen Welt**!

**Konstituierendes** Kennzeichen

- ▶ **rein funktionaler Programmierung**:
  - ▶ Die **völlige Abwesenheit** von **Seiteneffekten**!
- ▶ **Ein-/Ausgabe**:
  - ▶ Die **unvermeidbare Anwesenheit** von **Seiteneffekten**!  
Ein- und Ausgabe, lesen und schreiben verändern den Zustand der äußeren Welt **notwendig** und **irreversibel**.

Es gilt: **Ein-** und **Ausgabe** erzeugen (paradigmenunabhängig!) **notwendig** und **unvermeidbar Seiteneffekte** und sind **ohne Seiteneffekte undenkbar**.



# Ein-/Ausgabeverzicht ist keine Option

*“Der Benutzer lebt in der Zeit  
und kann nicht anders als zeitabhängig  
sein Programm beobachten.”*

Peter Pepper. *Funktionale Programmierung*.  
Springer-V., 2. Auflage, 2003.

Konsequenz: Wir können **abstrahieren**

- ▶ von der Arbeitsweise des **Rechners**
- ▶ nicht aber von der des **Benutzers**.

Die Aufnahme **dialog-** und **interaktionsorientierter** Ein-/Ausgabebehandlung ist unverzichtbar, bringt uns an die Nahtstelle

- ▶ von **reiner funktionaler** und **imperativer** Programmierung
- und erfordert, diese zu **überschreiten**.

# Kapitel 15.1.2

## Warum (naive) Einfachheit versagt

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Ein-/Ausgabeoperationen

...in **funktionaler Programmierung** müssen (wie **alle** Operationen und Funktionen)

- ▶ von **funktionalem Typ** sein,
- ▶ ein **Resultat** liefern.

Damit zu **klären**: Was können deren

- ▶ **Typ** und **Resultat** sein?

# Leseoperationen

...liefern stets einen Wert.

- Naheliegend: Den Wert der **gelesenen** Eingabe.

Am **Beispiel** einer Leseoperation für **ganze Zahlen**:

```
-- Zur Illustration: Kein gültiges Haskell!
```

```
READ_INT :: INT
```

```
READ_INT = << Lies "ganze Zahl"
```

```
    {- Der unvermeidbare Seiteneffekt, durch  
       den der Zustand der Welt irreversibel  
       verändert wird! -}
```

```
    und liefere deren Wert als Resultat.
```

```
    {- Das formal erforderliche und inhalt-  
       lich auch gewollte Ergebnis der Lese-  
       operation! -}
```

```
>>
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Schreiboperationen

...liefern stets einen Wert.

- ▶ **Naheliegend:** Nichts.
- ▶ **Hilfsweise:** (i) Den geschriebenen Wert, oder (ii) einen Wahrheitswert in Abhängigkeit des Erfolgs der Operation; oder (iii) irgendeinen Wert (beliebig; beliebig, aber fest).

Am **Bsp.** einer Schreibop. für **Zeichen** nach Vorschlag (i):

-- Zur Illustration: Kein gültiges Haskell!

PRINT\_STRING :: **STRING** -> **STRING**

PRINT\_STRING s =

<< **Gib** am Bildschirm den Wert von **s** **aus**

**{- Der unvermeidbare Seiteneffekt, durch den der  
Zustand der Welt irreversibel verändert wird! -}**

und **liefere** **s** als **Resultat**.

**{- Das formal erforderliche Ergebnis der Schreib-  
operation! -}**

>>

# Erstes Problem

Betrachte folgende einfache **interaktive Programmieraufgabe**:

- Schreibe ein Programm, dass (1) eine ganze Zahl liest und anschließend (2) einen frei wählbaren Text schreibt.

**Naheliegend:** Komponiere die beiden Funktionen `READ_INT` und `PRINT_STRING` sequentiell mittels **Funktionskomposition**:

$$\begin{aligned} (.) &:: (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C) \\ (F \ . \ G) \ X &= F \ (G \ X) \end{aligned}$$

Das ergibt:

`(PRINT_STRING . READ_INT)`

**Jedoch:** Die Komposition **scheitert**.

`READ_INT`        :: `INT`  
`PRINT_STRING` :: `STRING -> STRING`

...sind **nicht typkompatibel** für Komposition mittels `(.)`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Zweites Problem (1)

Betrachte folgendes Beispiel:

`KONSTANTE = 42 :: INT`

`FUN :: INT -> INT`

`FUN N = N + KONSTANTE`

`FUN' :: INT -> INT`

`FUN' N = N + READ_INT`

- ▶ Anders als der Wert von `FUN`, hängt der Wert von `FUN'` nicht allein vom Argumentwert, sondern auch vom Wert der Eingabeoperation ab.
- ▶ Das gilt auch für Funktionen, die sich direkt/indirekt auf `FUN'` abstützen: Die Programmbedeutung wird schwer durchschaubar.
- ▶ Jeder Aufruf von `FUN'` (u. sich darauf abstützender Funktionen) kann trotz gleichen Argumentwerts einen anderen Wert liefern. `FUN' 42 == FUN' 42` gilt **nicht** länger. `FUN'` und sich darauf abstützende 'Funktionen' werden **Relationen**.

# Zweites Problem (2)

Betrachte folgende Wertvereinbarungen:

WERT	= (17+4)*2 :: INT
DIFF	= WERT - WERT
DIFF'	= READ_INT - READ_INT
WAHR_ODER_FALSCH	= (DIFF + DIFF' == 0) :: BOOL

Ausdruck

- ▶ DIFF hat stets den Wert 0; gleich, ob WERT zunächst als linker oder rechter Operand der Differenz ausgewertet wird.
- ▶ DIFF' hat unterschiedliche Werte, wenn die Auswertung von linker und rechter Leseoperation vertauscht wird bei insgesamt gleichen (aber voneinander verschiedenen) eingelesenen Zahlen.
- ▶ WAHR\_ODER\_FALSCH hat abhängig von DIFF' den Wert TRUE oder FALSE, ist also nicht konstant.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Damit: Verlust referentieller Transparenz

Die Beispiele zeigen: **Ein-/Ausgabe** lösen das tragende Grundprinzip **reiner** funktionaler Programmierung auf:

- ▶ **Referentielle Transparenz**

...und sich daraus ergebende Gewissheiten:

- ▶ Keine Veränderungen des Zustands der äußeren Welt (**Seiteneffektfreiheit**).
- ▶ Der Wert eines Ausdrucks hängt nur vom Wert seiner Teilausdrücke ab (**Kompositionalität**), nicht von der Reihenfolge ihrer Auswertung (**Reihenfolgenunabhängigkeit**).
- ▶ Der Wert eines Ausdrucks ist unveränderlich über die Zeit (**Zeitunabhängigkeit**); er verändert sich nicht durch die Anzahl seiner Auswertungen (**Auswertungshäufigkeitsunabhängigkeit**).
- ▶ Ein Ausdruck darf stets durch seinen Wert ersetzt werden und umgekehrt (**Austauschbarkeit**).

# Ein-/Ausgabe

...stellt somit ein weiteres leitendes **Prinzip reiner funktionaler** (und allgemeiner **deklarativer**) Programmierung infrage:

- ▶ Die Betonung des **'was'** (die Ergebnisse) statt des **'wie'** (die Art ihrer Berechnung)

...rüttelt insgesamt an den **Grundfesten**, auf die sich

- ▶ **reine funktionale** Programmierung

gründet und von denen sich ihre

- ▶ **Stärke** und **Eleganz**

ableitet.

# Kapitel 15.2

## Haskells Lösung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 15.2.1

## Konzeption und Umsetzung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

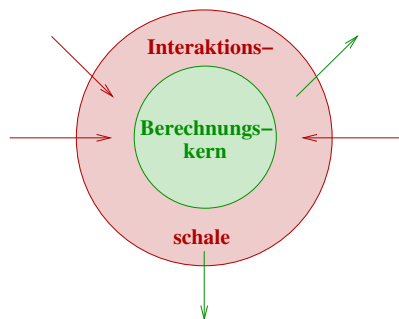
Kap. 12

Kap. 13

# Konzeptuelle E/A-Lösung Haskells

Konzeptuell wird in **Haskell** ein Programm geteilt in

- ▶ einen **rein funktionalen Berechnungskern**
- ▶ eine **imperativartige Dialog- und Interaktionsschale**.



Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson, 2004, S. 89.

# Haskells Umsetzung d. E/A-Lsg. im Überblick

**A)** Ein neuer (vordefinierter) Datentyp für Ein-/Ausgabe:

- ▶ `data IO a = ...` (Details implementierungsintern versteckt)

Vordefinierte primitive E/A-Operationen:

- ▶ `getChar :: IO Char`  
`getInt :: IO Int`  
...
- ▶ `putChar :: Char -> IO ()`  
`putInt :: Int -> IO ()`  
...

**B)** Ein Operator zur Komposition von E/A-Operationen:

- ▶ `(>>=) :: IO a -> (a -> IO b) -> IO b`
- ▶ 'Syntaktischer Zucker' für `(>>=)`: `do`-Notation.

**C)** Zwei Vermittlungsoperatoren:

- ▶ `return :: a -> IO a`
- ▶ `'<-' :: IO a -> a`

( $\rightsquigarrow$  informell!)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Lösungsbeiträge d. Umsetzungsbestandteile (1)

**A)** Trennung in rein funktionalen Berechnungskern und imperativartige Dialog- und Interaktionsschale:

Der Datentyp `(IO a)` erlaubt die Unterscheidung von Typen

- ▶ des rein funktionalen Berechnungskerns (`Char`, `Int`, `Bool`, etc.)
- ▶ der imperativartigen Dialog- und Interaktionsschale (`(IO Char)`), `(IO Int)`, `(IO Bool)`, etc.)

**Effekt:** `IO`-Werte können nicht das gesamte Programm 'kontaminieren'. Vereinbarungen wie für `wahr_oder_falsch` und `fun'` sind in Haskell typinkorrekt und nicht (mehr) möglich; sie werden vom Typsystem ausgeschlossen und abgewiesen:

```
fun' :: Int -> Int      wert = (17+4)*2 :: Int
fun' n = n + getInt     wahr_oder_falsch ((-)-inkompatibel)
      :: Int :: IO Int      = (wert - wert) + (getInt - getInt)
      ((+)-inkompatibel)   :: Int      :: IO Int :: IO Int
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Lösungsbeiträge d. Umsetzungsbestandteile (2)

**B)** Festlegung der zeitlichen Abfolge von E/A-Operationen  
(“Der Benutzer lebt in der Zeit...und kann nicht anders...”):

Der Kompositionsoperator ( $\gg=$ ) (oder gleichwertig die **do**-Notation, s. Kap. 15.4) erlauben die präzise Festlegung der

- ▶ zeitlichen Abfolge von E/A-Operationen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Lösungsbeiträge d. Umsetzungsbestandteile (3)

## C) Verbindung von funktionalem Kern und E/A-Schale

- ▶ **return**: Von Kern in Schale (in äußere Welt).
- ▶ **<-**: Von Schale (von äußerer Welt) in Kern.

### Informell:

- ▶ **return** erlaubt rein funktionale Werte (engl. pure values) aus dem funktionalen Kern über die Schale als seiten-effektverursachende Werte (engl. impure values) in die äußere Welt zu transferieren.  
... 'kontaminieren' reiner Werte.
- ▶ **<-** erlaubt den 'reinen' Anteil (a-Wert) seiteneffektverursachender Werte ((IO a)-Wert) aus der äußeren Welt in den funktionalen Kern zu transferieren.  
... 'dekontaminieren' E/A-verschmutzter Werte.

# Lösungsbeiträge d. Umsetzungsbestandteile (4)

**Kontaminierung**, **Dekontaminierung** noch bildhafter:

- ▶ **<-**: **Dekontaminierung** E/A-verschmutzter Werte  $\rightsquigarrow$  aus einem (IO a)-Wert wird ein a-Wert:

$\text{<-} :: \underbrace{\text{IO } a}_{\text{Schale}} \rightarrow \underbrace{a}_{\text{Kern}}$   
nach

**<-** nimmt einen E/A-verschmutzten (IO a)-Wert und extrahiert daraus den rein funktionalen sauberen a-Wert, der dadurch für den rein funktionalen Berechnungskern nutzbar wird.

- ▶ **return**: **Kontaminierung** rein funktionaler Werte  $\rightsquigarrow$  aus einem a-Wert wird ein (IO a)-Wert:

$\text{return} :: \underbrace{a}_{\text{Kern}} \rightarrow \underbrace{\text{IO } a}_{\text{Schale}}$   
nach

**return** nimmt einen rein funktionalen sauberen a-Wert und verschmutzt ihn zu einem (IO a)-Wert, der dadurch für und in der äußeren Welt nutzbar wird.

# Lösungsbeiträge d. Umsetzungsbestandteile (5)

**Bemerkung:** `return` und `<-` verhalten sich in diesem Sinne dual oder invers zueinander, wobei allerdings

- ▶ `return` eine (gewöhnliche) Funktion
- ▶ `<-` einen Wertvereinbarungsoperator (ähnlich `:=` oder `=`) aus imperativen, objektorientierten Sprachen bezeichnet, einen Wertvereinbarungsoperator mit integrierter Dekontaminationsfunktionalität: Dekontamination durch Auspacken, durch Extraktion des `a`-Werts aus einem (`IO a`)-Wert.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 15.2.2

## Aktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Aktionen: Ausdrücke vom Typ (IO a)

## Ausdrücke vom Typ (IO a)

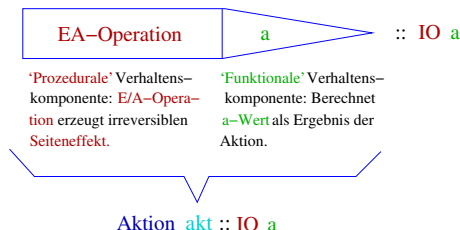
- ▶ sind **wertliefernde** ('funktionaler' Anteil) **E/A-Operationen** ('prozeduraler' Anteil).
- ▶ bewirken einen **Lese-** oder **Schreibseiteneffekt** (prozedurales Verhalten) **und** liefern einen **a-Wert** als Ergebnis (**funktionales** Verhalten), der eingepackt als **(IO a)**-Wert zur Verfügung gestellt wird.
- ▶ heißen **Aktionen** (oder **Kommandos**) (engl. **actions** oder **commands**).

## Informell:

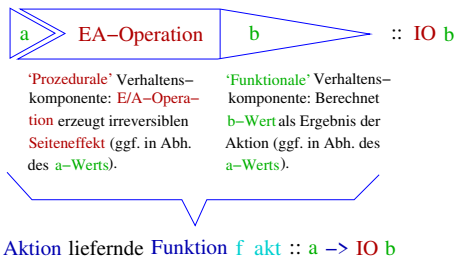
$$\begin{aligned}\text{Aktion} &= (1) \text{ E/A-Operation ('prozedural')} \\ &\quad + (2) \text{ Wertlieferung ('funktional')} \\ &= \text{wertliefernde E/A-Operation}\end{aligned}$$

# Veranschaulichung des Effekts von Aktionen

Aktion  $\text{akt} :: \text{IO } a$



Aktion liefernde Funktion  $\text{f\_akt} :: a \rightarrow \text{IO } b$



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

...aller **Leseaktionen** ist

- ▶ **(IO a)** (für 'lesegeeignete' Typinstanzen von **a**).

Der in einen **a**-Wert transformierte gelesene Wert wird als (formal erforderliches und inhaltlich gewolltes) Ergebnis von Leseoperationen verwendet.

...aller **Schreibaktionen** ist

- ▶ **(IO ())** mit **()** der einelementige **Nulltupeltyp** mit gleichbenanntem einzigen Datenwert **()**.

**()** als (einziger) Wert des Nulltupeltyps **()** wird als **formal erforderliches** Ergebnis von Schreiboperationen verwendet.

# Auswertung, Ausführung von Aktionen

Wegen des kombinierten

- ▶ **prozeduralen** (seiteneffekterzeugende Lese-/Schreiboperation) und
- ▶ **funktionalen** (Wert als Ergebnis liefernden)

Effekts der Auswertung von **Aktionen** (oder **E/A-Ausdrücken**), spricht man statt von **Auswertung** meist von **Ausführung** von **Aktionen** (oder **E/A-Ausdrücken**).



# Interpretation der Signatur von ( $\gg=$ )

...des Kompositionsoperators ( $\gg=$ ):

► ( $\gg=$ ) ::  $\text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b$

Die Signatur liefert:

- ( $\gg=$ ) ist eine Abbildung, die eine (Argument-) Aktion mit einem  $a$ -Wert als Ergebnis (d.h. einen  $(\text{IO } a)$ -Wert) auf eine (Bild-) Aktion mit einem  $b$ -Wert als Ergebnis abbildet (d.h. auf einen  $(\text{IO } b)$ -Wert) mithilfe einer Funktion, deren Ergebnis angewendet auf den  $a$ -Ergebniswert der Argumentaktion die gesuchte Bildaktion ist.

# Interpretation der Signatur von ( $\gg$ )

...des Kompositionsoperators ( $\gg$ ):

► ( $\gg$ ) :: IO a  $\rightarrow$  IO b  $\rightarrow$  IO b

Die Signatur liefert:

- ( $\gg$ ) ist eine Abbildung, die eine (Argument-) Aktion mit einem a-Wert als Ergebnis (d.h. einen (IO a)-Wert) und eine zweite (Argument-) Aktion mit einem b-Wert als Ergebnis (d.h. einen (IO b)-Wert) auf diese zweite Aktion als Bildaktion abbildet.

(Scheinbar hat das erste Argument keine Bedeutung und verschwindet; dies gilt für sein funktionales Ergebnis, den a-Wert, nicht aber für seinen prozeduralen Lese-/Schreibseiteneffekt!)

# Interpretation der Signatur von `return`

...der aktionsliefernden Funktion `return`:

► `return :: a -> IO a`

Die **Signatur** liefert:

- `return` ist eine Abbildung, die einen `a`-Wert auf eine Aktion mit einem `a`-Wert als Ergebnis abbildet (d.h. auf einen `(IO a)`-Wert).

# Operationelle Bedeutung

...des Kompositionsoperators ( $\gg=$ ):

►  $(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$

Sei  $(akt :: IO\ a)$  eine Aktion,  $(f\_akt :: a \rightarrow IO\ b)$  eine Aktion liefernde Abbildung.

Operationelle Bedeutung der Komposition  $(akt \gg= f\_akt)$ :

- $akt$  wird ausgeführt, bewirkt dabei einen Lese- oder Schreibseiteneffekt und liefert als Ergebnis einen  $a$ -Wert; dieser  $a$ -Wert wird zum Argument von  $f\_akt$ , deren Bildwert vom Typ  $(IO\ b)$  eine Aktion ist, die ausgeführt wird, dabei einen weiteren Lese- oder Schreibseiteneffekt bewirkt und als Ergebnis einen  $b$ -Wert liefert; dieser ist zugleich das (funktionale) Ergebnis der Komposition  $(akt \gg= f\_akt)$ .

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

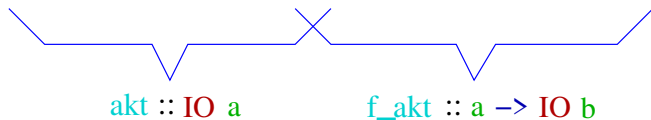
Teil V

Kap. 12

Kap. 13

# Veranschaulichung der operat. Bedeutung

...der komponierten Aktion ( $\text{akt} \gg= \text{f\_akt}$ ):



$$\text{akt} \gg= \text{f\_akt} \quad \hat{=} \quad \text{akt} \gg= \backslash x \rightarrow \text{f\_akt } x$$

# Operationelle Bedeutung

...des Kompositionsoperators ( $\gg$ ):

► ( $\gg$ ) :: IO a  $\rightarrow$  IO b  $\rightarrow$  IO b

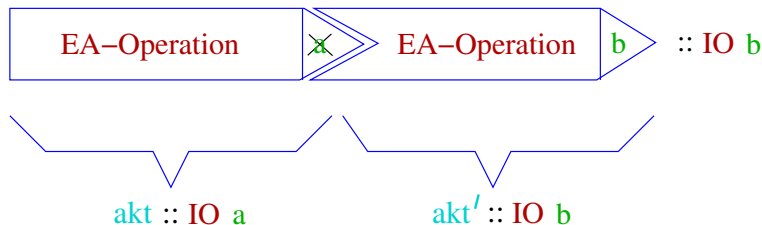
Seien ( $\text{akt} :: \text{IO a}$ ), ( $\text{akt}' :: \text{IO b}$ ) zwei Aktionen.

Operationelle Bedeutung der Komposition: ( $\text{akt} \gg \text{akt}'$ ):

- $\text{akt}$  wird ausgeführt, bewirkt dabei einen Lese- oder Schreibseiteneffekt und liefert als Ergebnis einen  $\text{a}$ -Wert. Dieser  $\text{a}$ -Wert wird ignoriert und unmittelbar die Aktion  $\text{akt}'$  ausgeführt, die dabei einen weiteren Lese- oder Schreibseiteneffekt bewirkt und als Ergebnis einen  $\text{b}$ -Wert liefert; dieser ist zugleich das (funktionale) Ergebnis der Komposition ( $\text{akt} \gg \text{akt}'$ ).

# Veranschaulichung der operat. Bedeutung

...der komponierten Aktion ( $\text{akt} \gg \text{akt}'$ ):



$$\text{akt} \gg \text{akt}' \hat{=} \text{akt} \gg= \_ \rightarrow \text{akt}'$$

# Komposition: 'binde-dann'-, 'dann'-Operator

## Die Kompositionsoperatoren

- ▶  $(>>=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$
- ▶  $(>>) :: IO\ a \rightarrow IO\ b \rightarrow IO\ b$   
 $akt >> akt' = akt >>= \_ \rightarrow akt'$  (vordefiniert)

...gelesen als

- ▶ binde-dann-Operator (engl. `bind` oder `then`)
- ▶ dann-Operator (engl. `sequence`).

**Bem.:** Die Definition von  $(>>)$  macht deutlich, dass  $(>>)$  kein eigenständiger Operator, sondern von  $(>>=)$  abgeleitet und eine spezielle Anwendung von  $(>>=)$  ist, die das Ergebnis von `akt` (`a`-Wert) als Argument für `akt'` (`\_ -> akt'`) ignoriert: Der `a`-Wert von `akt` wird anders als bei  $(>>=)$  nicht für weitere Verwendung an einen Namen gebunden, er wird 'vergessen'.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Operationelle Bedeutung

...der Funktion `return`:

► `return :: a -> IO a`

Sei (`w :: a`) ein `a`-Wert.

Operationelle Bedeutung des aktionsliefernden Ausdrucks  
(`return w`):

- `return` bildet den `a`-Wert `w` in 'offensichtlicher' Weise auf den 'entsprechenden' (`IO a`)-Wert ab, *ohne* einen Lese- oder Schreibseiteneffekt zu bewirken.

(Das **prozedurale** Verhalten von `return` entspricht der leeren Anweisung '*skip*'; `return` hat (deshalb) abweichend von anderen Aktionen nur ein **funktionales** beobachtbares Verhalten, kein prozedurales).

# Veranschaulichung

...der operationellen Bedeutung von `return`:



**'Prozedurale'** Verhaltenskomponente: 'Leer'; keine E/A-Operation, kein Seiteneffekt.

**'Funktionale'** Verhaltenskomponente: Reicht den `a-Wert` als Ergebnis der Aktion durch.

Aktion `return` `:: a -> IO a`

# Wichtig zu beachten

Die **E/A**-Aktion `return` in Haskell

- ▶ hat eine gänzlich andere Aufgabe und Bedeutung als das aus imperativen oder objektorientierten Sprachen bekannte `return`; außer der Namensgleichheit besteht weder konzeptuell noch funktionell eine Ähnlichkeit.
- ▶ Haskell's `return` kann in einer Aktionssequenz auftreten und ausgewertet werden, ohne dass dadurch die Auswertung der restlichen Aktionssequenz beendet würde; `return` kann deshalb auch mehrfach in sinnvoller Weise in einer Aktionssequenz auftreten.
- ▶ Zum Verständnis von Haskell's `return` ist eine Orientierung am imperativen, objektorientierten `return` deshalb nicht sinnvoll und allenfalls irreführend.

# Kapitel 15.2.3

## Aktionssequenzen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Die Kompositionsoperatoren ( $\gg=$ ) und ( $\gg$ )

...erlauben die Bildung (assoziativer) Aktionssequenzen:

```
akt1 >>= f_akt2 >> akt3 >> akt4 >>= f_akt5 >>= return f
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Allgemeines Muster von Aktionssequenzen

...mit ( $\gg=$ ) vom Typ ( $\text{IO } b$ ):

```
akt1 >>= \p1 ->
akt2 >>= \p2 ->
...
aktn >>= \pn ->
return (f p1 p2 ... pn)
```

-- p für Parameter

mit

```
f :: a1 -> a2 -> ... -> an -> b
```

geeigneter Verknüpfungsoperation und

```
akt1 :: IO a1
akt2 :: IO a2
...
aktn :: IO an
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Aktionssequenzen mit ( $\gg=$ ) und ( $\gg$ )

...mit und ohne Rückführung von ( $\gg$ ) auf ( $\gg=$ ):

akt1 $\gg=$ \p1 ->	akt1 $\gg=$ \p1 ->
akt2 $\gg=$ \_ ->	akt2 $\gg$
akt3 $\gg=$ \_ ->	akt3 $\gg$
akt4 $\gg=$ \p4 ->	akt4 $\gg=$ \p4 ->
...	...
aktn $\gg=$ \pn ->	aktn $\gg=$ \pn ->
return (f p1 p4 ... pn)	return (f p1 p4 ... pn)

...der **Typ** einer **Aktionssequenz** ist durch den **Typ** der **letzten Aktion** bestimmt.

# Schrittweise Aktionssequenzauswertung (1)

```
akt1 >>= \p1 ->
```

```
akt2 >>= \p2 ->
```

```
akt3 >>= \p3 ->
```

```
...
```

```
aktn >>= \pn ->
```

```
return (f p1 p2 p3 ... pn)
```

```
->> (Aktion akt1 erzeugt E/A-Effekt und liefert Wert w1)
```

```
(\p1 ->
```

```
  akt2 >>= \p2 ->
```

```
  akt3 >>= \p3 ->
```

```
  ...
```

```
  aktn >>= \pn ->
```

```
  return (f p1 p2 p3 ... pn) ) w1
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Schrittweise Auswertung Aktionssequenz (2)

```
->>(Aktion akt2 erzeugt E/A-Effekt und liefert Wert w2)
```

```
(\p1 ->  
  (\p2 ->  
    akt3 >>= \p3 ->  
    ...  
    aktn >>= \pn ->  
    return (f p1 p2 p3 ... pn)) w1) w2
```

```
->> (Aktion akt3 erzeugt E/A-Effekt und liefert Wert w3)  
(...
```

```
->> (Aktion aktn erzeugt E/A-Effekt und liefert Wert wn)
```

```
(\p1 ->  
  (\p2 ->  
    (\p3 ->  
      (...  
        (\pn ->  
          return (f p1 p2 p3 ... pn)) w1) w2) w3)...)) wn
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Schrittweise Aktionssequenzauswertung (3)

->> (Applikation der 1-ten Funktion auf w1)

```
(\p2 ->  
  (\p3 ->  
    (...  
      (\pn ->  
        return (f w1 p2 p3 ... pn)) w2) w3)... ) wn
```

->> (Applikation der 2-ten Funktion auf w2)

```
(\p3 ->  
  (...  
    (\pn ->  
      return (f w1 w2 p3 ... pn)) w3)... ) wn
```

->> (Applikation der 3-ten Funktion auf w3)

...

->> (Applikation der n-ten Funktion auf wn)

```
return (f w1 w2 w3 ... wn)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Schrittweise Aktionssequenzauswertung (4)

->> (Linksassoziative Klammerung von (f w1 w2 w3 ...wn))

return ((...(((f w1) w2) w3) ...) wn))

->> (Anwendung von f auf w1 w2 w3 ... wn liefert b-Wert w,  
d.h.: (...(((f w1) w2) w3) ...) wn) ->> w :: b)

return w

->> (Anwendung von return auf w liefert (ohne E/A-Effekt)  
das Ergebnis erg vom Typ IO b der Aktionssequenz,  
d.h. erg = IO w :: IO b)

erg = IO w :: IO b  
Datenkonstruktor Typkonstruktor

...in Kapitel 15.4 werden wir Haskell's do-Notation als suggestivere und bequemere Schreibweise für Aktionssequenzen kennenlernen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17

# Kapitel 15.2.4

## Zur Sonderstellung des Typs (IO a)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Zum Unterschied von (IO a) und (MT a) (1)

Vergleiche die 'gewöhnliche' Typdeklaration und Wertvereinbarung von:

```
data MT a = MT a deriving Show    -- MT für 'MeinTyp'
z = MT 'u' :: MT Char             -- z für 'zeichen'
```

mit denjenigen der folgenden E/A-Aktionen:

```
data IO a = IO ...                -- Details impl.-intern
akt  = getChar :: IO Char
akt' = putChar 'v' :: IO ()
akt'' = putChar :: Char -> IO ()
akt''' = return 'w' :: IO Char
akt'''' = return :: a -> IO a
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## Zum Unterschied von (IO a) und (MT a) (2)

Die **Auswertung** von **Ausdruck z** bewirkt:

- ▶ Das Zeichen 'u' (eingepackt in den Datenwertkonstruktor **MT**) wird geliefert (**funktionales Verhalten**); darüber hinaus passiert nichts, kein zusätzliches, insbesondere kein prozedurales Verhalten.

# Zum Unterschied von (IO a) und (MT a) (3)

## Die Auswertung von Aktion

- ▶ **akt** bewirkt:
  - (1) Ein Zeichen wird vom Bildschirm gelesen (**prozedurale E-Operation**) und
  - (2) der Wert des gelesenen Zeichens wird als **Ergebnis** (eingepackt in den Datenwertkonstruktor **IO**) geliefert (**funktionales Verhalten**).
- ▶ **akt'** bewirkt:
  - (1) Das Zeichen '**v**' wird auf den Bildschirm geschrieben (**prozedurale A-Operation**) und
  - (2) der Wert **()** des Nulltupeltyps **()** wird als Ergebnis von **akt'** (eingepackt in den Datenwertkonstruktor **IO**) geliefert (**funktionales Verhalten**).
- ▶ (**akt''** '**v**') bewirkt: Ident zur Auswertung von **akt'**.

# Zum Unterschied von (IO a) und (MT a) (4)

## Die Auswertung von Aktion

- ▶ `akt'''` bewirkt:
  - (1) Ohne dass eine E/A-Operation stattfindet (das prozedurale Verhalten ist 'leer', entsprechend '*skip*') wird
  - (2) das Zeichen '`w`' als Ergebnis von `akt'''` (eingepackt in den Datenwertkonstruktor `IO`) geliefert (`funktionales Verhalten`).
- ▶ `akt''''` bewirkt: Fehlschlag; `akt''''` = `return` vereinbart einen Aliasnamen für die Funktion (genauer: Aktion) `return`; ohne Argument lassen sich Funktionen nicht auswerten (vgl. auch `akt''`).
- ▶ (`akt'''' 'w'`) bewirkt: Ident zur Auswertung von `akt'''`.



# Zusammenfassung (1)

Die äußere Ähnlichkeit der Deklarationen

```
data MT a = MT a                -- rein fkt. Typ
z = MT 'u' :: MT Char          -- rein fkt. Ausdruck

data IO a = IO ...              -- E/A-Typ
akt = getChar :: IO Char       -- E/A-Ausd./Aktion
akt' = putChar 'v' :: IO ()    -- E/A-Ausd./Aktion
```

ist oberflächlich: Die Auswertung

- ▶ rein funktionaler Ausdrücke wie `z`

ist wesentlich anders als die von

- ▶ E/A-Ausdrücken (oder Aktionen) wie `akt`, `akt'`.

# Zusammenfassung (2)

## Auswertung eines

- ▶ **rein funktionalen** Ausdrucks (engl. **pure** expression): Der Wert des Ausdrucks wird geliefert ('**funktionaler**' Effekt), sonst (passiert) nichts.
- ▶ **E/A**-Ausdrucks (engl. **impure** expression):
  - (1) Eine **E/A**- Operation wird ausgeführt (Lese-/Schreibseiteneffekt wird generiert, '**prozeduraler**' Effekt).
  - (2) ein **a**-Wert (eingepackt in den Datenwertkonstruktor **IO**) wird als Wert des **E/A**-Ausdrucks geliefert ('**funktionaler**' Effekt).

**Ausnahme:** Der **E/A**-Ausdruck (**return** **ausd** :: **IO T**) für (**ausd** :: **T**) und **T** konkreter Typ liefert den Wert des Ausdrucks **ausd** (eingepackt in den Datenwertkonstruktor **IO**) als Ergebnis **ohne** eine **E/A**-Operation auszuführen (und somit ohne einen Lese-/Schreibseiteneffekt zu bewirken).

# Kapitel 15.3

## E/A-Operationen, E/A-Sequenzen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Vordefinierte Ein-/Ausgabeoperationen

...zum Lesen und Schreiben vom bzw. auf den Bildschirm.

## Leseoperationen:

```
getChar  :: IO Char
getInt   :: IO Int
getline  :: IO String
readIO   :: Read a => String -> IO a
readLn   :: Read a => IO a
...
```

## Schreiboperationen:

```
putChar   :: Char    -> IO ()
putStr    :: String  -> IO ()
putStrLn  :: String  -> IO ()
print     :: Show a  => a  -> IO ()
...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Vordefinierte Ein-/Ausgabeoperationen

...zum Lesen und Schreiben aus bzw. in Dateien.

Leseoperationen:

```
readFile :: FilePath -> IO String
```

Schreiboperationen:

```
writeFile :: FilePath -> String -> IO ()
```

```
appendFile :: FilePath -> String -> IO ()
```

Dateiende-Prädikat:

```
isEOF :: FilePath -> Bool
```

Pfad-/Dateinamen:

```
type FilePath = String
```

...mit betriebssystemabhängigen Werten von `FilePath`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Beispiel: Anwendungen v. `putStrLn`, `putStr`

...für das 'Hallo, Welt'-Programm:

```
halloWelt :: IO ()  
halloWelt = putStrLn "Hallo, Welt!"
```

...mit `putStrLn` zur Ausgabe einer Zeichenreihe mit anschließendem Zeilenumbruch:

```
putStrLn :: String -> IO ()  
putStrLn = putStr . (++ "\n")
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17

# E/A-Operationen und die Fkt. show, read

Mithilfe der Funktion `show` der Typklasse `Show` und der globalen (engl. top level) Funktion `read` (`read` keine Funktion der Typklasse `Read`!):

```
show :: Show a => a -> String
```

...lassen sich Werte von Instanztypen der Typklasse `Show` ausgeben und von Instanztypen der Typklasse `Read` einlesen:

```
putLine :: Show a => a -> IO ()
```

```
putLine = putStrLn . show
```

```
print :: Show a => a -> IO ()
```

```
print = putLine
```

```
read :: Read a => String -> a
```

```
read s = ...                -- definiert im Präludium
```

**Bem.:** Vordefinierte Instanzen von `Show` und `Read`: Alle im Präludium definierten Typen mit Ausnahme v. Funktions- u. `IO`-Typen.

# E/A-Sequenzen mittels (.) und (>>=)

...mittels Funktionskomposition (.).

Schreiben mit Zeilenvorschub (vordefinierte Sequenz):

```
putStrLn :: String -> IO ()  
putStrLn = putStr . (++ "\n")
```

..mittels IO-Komposition (>>=).

Lesen einer Zeile und anschließendes Schreiben der gelesenen Zeile (selbstdefinierte Sequenz):

```
echo :: IO ()  
echo = getLine >>= (\zeile -> putLine zeile)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Kapitel 15.4

## Die do-Notation

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Die do-Notation

...‘syntaktischer Zucker’ für die **IO**-Kompositionsoperatoren ( $\gg=$ ) und ( $\gg$ ) zur gefälligeren, **imperativähnlicheren**

- Bildung von **Ein-/Ausgabesequenzen**.

Zwei **Beispiele**:

```
do zeile <- getLine statt getLine >>= (\zeile ->
    putStrLn zeile                                putStrLn zeile)
```

```
do putStr "fun"      statt putStr "fun" >> putStr "\n"
   putStr "\n"       oder putStr "fun" >>= (\_ ->
                                           putStr "\n")
```

**Bem.:** Ein **do**-Ausdruck entspricht semantisch einer **Sequenz** von **E/A-Operationen** und kann (deshalb) auf eine beliebige Anzahl von **Aktionen** als Argumente angewendet werden (in den obigen beiden Beispielen jeweils zwei).

Die **Abseitsregel** gilt auch in **do**-Ausdrücken.

# Allgemeines Muster von do-Ausdrücken

```
do w1 <- akt1      -- Sprechweise: akti Generator
   w2 <- akt2      -- für Wert wi vom Typ ai
   ...
   wn <- aktn
   return (f w1 w2 ... wn)
```

mit

```
f :: a1 -> a2 -> ... -> an -> b
```

geeigneter Verknüpfungsfunktion und

```
akt1 :: IO a1
akt2 :: IO a2
...
aktn :: IO an
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# do-Ausdrücke in einer Zeile

...ein **do**-Ausdruck wie

```
do w1 <- akt1  
   w2 <- akt2  
   ...  
   wn <- aktn  
return (f w1 w2 ... wn)
```

lässt sich (so gewünscht) mittels **';** bedeutungsgleich in einer Zeile schreiben:

```
do w1 <- akt1; ... ; wn <- aktn; return (f w1 w2 ... wn)
```

# Der Typ von do-Ausdrücken

...ist durch den **Typ** der **letzten Aktion** bestimmt:

```
( do w1 <- akt1
  w2 <- akt2
  ...
  wn <- aktn
  return (f w1 w2 ... wn) ) :: IO b
```

bzw.

```
( do w1 <- akt1; ...; wn <- aktn; return (f w1...wn) ) :: IO b
```

wobei jeweils

```
f :: a1 -> a2 -> ... -> an -> b
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Nicht verwendete Aktionsergebnisse

...in **do**-Ausdrücken.

**Aktionen** liefern stets ein **Ergebnis**. Bleibt es unverwendet (entspricht Aktionskomposition mit **(>>)** statt mit **(>>=)**), kann die Nichtverwendung syntaktisch dadurch ausgedrückt werden, dass ein Aktionsergebnis nicht an einen Wertnamen **wi**, sondern an **\_** 'gebunden' wird, quasi 'unbenannt' gebunden wird:

```
do w1 <- akt1
  _ <- akt2
  _ <- akt3
  w4 <- akt4
  ...
  wn <- aktn
return (f w1 w4 ... wn)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Weglassen unbenannter Bindungen

...noch einfacher können diese 'unbenannten' Bindungen auch ganz entfallen:

```
do w1 <- akt1
  akt2
  akt3
  w4 <- akt4
  ...
  wn <- aktn
return (f w1 w4 ... wn)
```

# Entsprechung von do- und (>>=)-Notation

Der **do**-Ausdruck

```
do w1 <- akt1
   w2 <- akt2
   ...
   wn <- aktn
return (f w1 w2 ... wn)
```

ist durch den **(>>=)**-Ausdruck definiert:

```
akt1 >>= \p1 ->
akt2 >>= \p2 ->
...
aktn >>= \pn ->
return (f p1 p2 ... pn)
```

Beide Ausdrücke haben dieselbe Bedeutung und entsprechen einander.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# A-Sequenzen mittels do-Notation

Einmaliges Schreiben einer Zeichenreihe mit Zeilenvorschub:

```
putStrLn :: String -> IO ()           -- Definition aus
putStrLn str = do putStrLn str         -- Präludium
                putStrLn "\n"
```

Zweimaliges Schreiben einer Zeichenreihe (mit Z-Vorschüben):

```
putStrLn_2mal :: String -> IO ()
putStrLn_2mal str = do putStrLn str
                    putStrLn str
```

Viermaliges Schreiben einer Zeichenreihe (mit Z-Vorschüben):

```
putStrLn_4mal :: String -> IO ()
putStrLn_4mal str = do putStrLn str
                        putStrLn str
                        putStrLn str
                        putStrLn str
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# E/A-Sequenzen mittels do-Notation

Zwei Lese-, eine Schreibaktion:

```
read2lines_and_report :: IO ()
read2lines_and_report
= do getLine      -- Z. wird gelesen u. vergessen
     getLine      -- Z. wird gelesen u. vergessen
     putStrLn "Zwei Zeilen gelesen."
```

Eine Lese-, zwei Schreibaktionen:

```
read1line_and_echo2times :: IO ()
read1line_and_echo2times
= do line <- getLine -- Z. w. gelesen u. gemerkt
     putStrLn line   -- Gemarkte Z. w. geschrieben
     putStrLn line   -- Gemarkte Z. w. geschrieben
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# A-Sequenzen parametrisierter Länge

n-maliges Schreiben einer Zeichenreihe (mit Z-Vorschüben):

```
putStrLn_nmal :: Int -> String -> IO ()
putStrLn_nmal n str
  = if n <= 1
    then putStrLn str
    else do putStrLn str
           putStrLn_nmal (n-1) str  -- Rekursion!
```

Das erlaubt auch folgende (alternative) Definitionen:

```
putStrLn_2mal :: String -> IO ()
putStrLn_2mal = putStrLn_nmal 2

putStrLn_4mal :: String -> IO ()
putStrLn_4mal = putStrLn_nmal 4
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# do-Ausdrücke mit return (1)

Lesen einer Zeichenreihe vom Bildschirm und Konversion in eine ganze Zahl:

```
getInt :: IO Int
getInt = do line <- getLine
          return (read line :: Int)
```

Im Detail:

```
getInt :: IO Int
getInt = do line <- getLine
          :: String      :: IO String
          return (read line :: Int)
                Konvertierung 'String' (der
                Typ von line) zu 'Int' (der
                Argumenttyp von return)
          :: IO Int
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## do-Ausdrücke mit return (2)

Bestimmung der Länge, der Zeichenzahl einer Datei:

```
groesse :: IO Int
groesse = do putStrLn "Dateiname = "
             name <- getLine
             text <- readFile name
             return (length text)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## do-Ausdrücke mit return (3)

Mit detaillierter Typinformation:

```
groesse :: IO Int
groesse = do putStrLn "Dateiname = "
           name  <-    getLine
           :: String      :: IO String
           text  <-    readFile name
           :: String      :: IO String
           return (length text)
                   :: String
                   :: Int
           :: IO Int
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 15.5

## Beispiele ausgewählter E/A-Programme

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 15.5.1

## Dialog- und Interaktionsprogramme

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Dialog- und Interaktionsprogramme

Zwei Frage/Antwort-Interaktionen mit dem Benutzer:

```
ask :: String -> IO String
ask frage = do putStrLn frage
              getLine

interAct :: IO ()           -- Bildschirm-Interaktion
interAct
  = do name <- ask "Wie heißen Sie?"
        putStrLn ("Willkommen " ++ name ++ "!")

interAct' :: IO ()          -- Datei-Interaktion
interAct'
  = do putStr "Bitte Dateinamen angeben: "
        dateiname <- getLine
        inhalt     <- readFile dateiname
        putStr inhalt
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Bedeutungsgleichheit von ( $\gg=$ ), ( $\gg$ ) und `do`

...für die Konstruktion von E/A-Sequenzen.

Die A-Sequenz mittels ( $\gg$ ):

```
writeFile "meineDatei.txt" "Hallo, Dateisystem!"  
>> putStr "Hallo, Welt!"
```

...ist bedeutungsgleich zur A-Sequenz mittels `do` (genauer: deren Bedeutung sie definiert):

```
do writeFile "meineDatei.txt" "Hallo, Dateisystem!"  
  putStr "Hallo, Welt!"
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Bedeutungsgleichheit von ( $\gg=$ ), ( $\gg$ ) und `do`

Die E/A-Sequenz mittels ( $\gg=$ ) und ( $\gg$ ):

```
incrementInt :: IO ()  
incrementInt  
  = getLine >>=  
    \zeile -> putStrLn (show (1+read zeile :: Int))
```

...ist bedeutungsgleich zur E/A-Sequenz mittels `do` (genauer: deren Bedeutung sie definiert):

```
incrementInt' :: IO ()  
incrementInt'  
  = do zeile <- getLine  
      putStrLn (show (1 + read zeile :: Int))
```

Informell: '`do`' entspricht ' $(\gg=)$  plus anonyme  $\lambda$ -Abstraktion'.

# Bedeutungsgleichh. ( $\gg=$ ), ( $\gg$ ) u. `do`, `return`

Die E-Sequenz mittels ( $\gg=$ ):

```
readStringPair :: IO (String,String)
readStringPair
  = getLine >>=
    (\zeile -> (getLine >>=
      (\zeile' -> (return (zeile,zeile')))))
```

...ist bedeutungsgleich zur E-Sequenz mittels `do` und `return`:

```
readStringPair' :: IO (String,String)
readStringPair'
  = do zeile <- getLine
      zeile' <- getLine
      return (zeile,zeile')
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Lokale Deklarationen in do-Ausdrücken

Die E/A-Sequenz (ohne lokale Deklarationen):

```
reverse2lines :: IO ()
reverse2lines
  = do line1 <- getLine
      line2 <- getLine
      putStrLn (reverse line2)
      putStrLn (reverse line1)
```

...ist bedeutungsgleich zur Sequenz mit lokalen Deklarationen:

```
reverse2lines :: IO ()
reverse2lines
  = do line1 <- getLine
      line2 <- getLine
      let rev1 = reverse line1
      let rev2 = reverse line2
      putStrLn rev2
      putStrLn rev1
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Unterschiedliche Bindung von `<-` und `let`

## Benannte Wertvereinbarungen mittels

- ▶ `<-`: für den `a`-Wert von **Aktionen** vom Typ `(IO a)` (für 'unreine' Werte aus der **äußeren Welt!**).
- ▶ `let`: für den Wert **rein funktionaler Ausdrücke** (für 'reine' Werte aus dem **rein funktionalen Programmkernel**).

# Kapitel 15.5.2

## Rekursive E/A-Programme

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Rekursive E/A-Programme (1)

...lesen und schreiben gelesener Eingaben: Kopieren.

Nichtterminierendes Kopieren (Notabbruch mit Ctrl-c):

```
kopiere :: IO ()
kopiere
  = do zeile <- getLine
      putStrLn zeile
      kopiere           -- Rekursion!
```

n-maliges Kopieren:

```
kopiere_n_mal :: Int -> IO ()
kopiere_n_mal n
  = if n <= 0
      then return ()
      else do zeile <- getLine
              putStrLn zeile
              kopiere_n_mal (n-1)           -- Rekursion!
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Rekursive E/A-Programme (2)

Kopieren bis zur Eingabe der leeren Zeile:

```
kopiere_bis_leer :: IO ()
kopiere_bis_leer
  = do zeile <- getLine
      if zeile == ""
      then return ()
      else do putStrLn zeile
              kopiere_bis_leer           -- Rekursion!
```

Kopieren bis zur Eingabe der leeren Zeile unter Mitzählen:

```
kopiere_bis_leer_und_zaehle_mit :: Int -> IO ()
kopiere_bis_leer_und_zaehle_mit n
  = do zeile <- getLine
      if zeile == ""
      then putStrLn
           (show n ++ " Zeilen gelesen u. kopiert.")
      else do putStrLn zeile
              kopiere_bis_leer_und_zaehle_mit (n+1)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17

# Rekursive E/A-Programme (3)

Summieren einer Folge ganzer Zahlen bis 0 eingegeben wird:

```
summiere :: IO Int
summiere
  = do n <- getInt
      if n == 0
      then return 0
      else (do m <- summiere
              return (n + m))
```

Vergleiche `summiere` mit:

```
sum :: [Int] -> Int
sum [] = 0
sum (n:ns)
  = let m = sum ns
      in (n + m)
```

```
sum' :: [Int] -> Int
sum' [] = 0
sum' (n:ns)
  = n + sum' ns
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Rekursive E/A-Programme (4)

Interaktives Summieren einer Folge ganzer Zahlen bis 0 eingegeben wird, abgestützt auf `summiere`:

```
summiere_interaktiv :: IO ()
summiere_interaktiv
  = do putStrLn "Gib ganze Zahl ein, je eine pro"
       putStrLn "Zeile. Diese werden summiert bis"
       putStrLn "Null eingegeben wird."
       summe <- summiere
       putStr "Der Summenwert ist "
       putLine summe
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 15.5.3

## Iterativartige E/A-Programme

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Iterativartige E/A-Programme

Iterativartiger Ausdruck/Programm, genauer die iterativartige Funktion `while`:

```
while :: IO Bool -> IO () -> IO ()
while bedingung aktion
  = do b <- bedingung
      if b
      then
        do aktion
           while bedingung aktion -- Rekursion!
      else
        return ()
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Zur operationellen Bedeutung der Fkt. while

## Intuitiv:

- ▶ Ist die Bedingung (`bedingung :: IO Bool`) erfüllt (und hat `b` somit den Wert `True`), so wird die Aktion (`aktion :: IO ()`) ausgeführt (do-Ausdruck im then-Ausdruck); anderenfalls endet die Ausführung/-wertung von `while` ohne weiteren E/A-Seiteneffekt mit dem Resultatwert `() :: ()`.
- ▶ Nach abgeschlossener Ausführung/-wertung von `aktion` (im Fall der erfüllten Bedingung) wird `while` rekursiv aufgerufen, wodurch insgesamt die 'iterativartige' Anmutung entsteht, dass eine Aktion so lange ausgeführt wird, wie eine Bedingung erfüllt ist.
- ▶ Mögliches Argument für die Bedingung: Der Ausdruck `isEOF :: IO Bool` zum Test auf das Eingabeende.

# Anwendung der Funktion `while`

...um eine Datei zeilenweise zu lesen und gelesene Zeilen wieder auszugeben, bis das Dateiende erreicht ist.

```
kopiere_eingabe_nach_ausgabe :: IO ()
kopiere_eingabe_nach_ausgabe
  = while (do wert <- isEOF      -- Arg. f. Param.
           return (not wert))   -- bedingung
         (do zeile <- getLine    -- Arg. f. Param.
           putStrLn zeile)      -- aktion
```

Bem.: Die Klammerung der Argumente von `while` ist nötig.

# Kapitel 15.5.4

## 'Iteration' vs. Rekursion

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Wertvereinbarung vs. Wertzuweisung

...funktionale Wertvereinbarung vs. imperative Wertzuweisung.

Zur Natur des

- ▶ Wertvereinbarungsoperators ' $<-$ ' in **do**-Ausdrücken

im Vergleich zum

- ▶ destruktiven Wertzuweisungsoperator ' $:=$ ' in destruktiven Zuweisung(sanweisung)en (engl. destructive assignments) imperativer Sprachen.

Tatsächlich besitzt

- ▶ ' $<-$ ' Ähnlichkeit mit einer Wertzuweisung, ist aber **gänzlich verschieden** der destruktiven Wertzuweisung imperativer Sprachen.

# Einmal-Wertvereinbarungsoperator '<-'

'<-' leistet eine **Einmal-Wertvereinbarung** für einen **Namen**:

- ▶ `zeile <- getLine` bindet das Resultat von `getLine` (allgemeiner: einer Eingabeoperation), an einen Namen, hier `zeile`.
- ▶ Diese **Verbindung** zwischen dem **Namen**, hier `zeile`, und dem von einer Eingabeoperation gelieferten **Wert**, hier `getLine`, bleibt für den gesamten Programmablauf erhalten und ist **nicht** mehr **veränderbar**.

# Mehrfach-Wertzuweisungsoperator ‘:=’

‘:=’ leistet eine temporäre Wertzuweisung an eine durch einen Namen bezeichnete Speicherzelle:

- ▶ **x := READ\_STRING**: Der von **READ\_STRING** gelesene Wert wird in die von **x** bezeichnete Speicherzelle geschrieben; der vorher dort gespeicherte Wert wird dabei überschrieben und zerstört (**destruktiv!**).
- ▶ Die durch die Zuweisung geschaffene Verbindung zwischen Name (d.h. der mit ihm bezeichneten Speicherzelle) und Wert (d.h. dem Inhalt der Speicherzelle) bleibt so lange erhalten (**temporär!**), bis sie durch eine erneute Zuweisung an diese Zelle überschrieben und zerstört wird (**destruktive Zuweisung!**).
- ▶ Der Inhalt einer Speicherzelle kann jederzeit und beliebig oft überschrieben werden und so die Verbindung von Name und Wert geändert werden (s. a. Anhang A.6).

# Zur Wirkung von Einmal-Wertvereinbarungen

...anhand eines **Beispiels**:

**Aufgabe:** Schreibe ein Programm, das so lange eine Zeile vom Bildschirm einliest und wieder ausgibt, bis schließlich die leere Zeile eingelesen wird und die Ausführung abgebrochen wird.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Der Effekt von Einmal-Wertvereinbarungen

‘Iterativer’ Lösungsversuch mittels **while**-Funktion/Ausdrucks:

```
goUntilEmpty :: IO ()
goUntilEmpty
  = do zeile <- getLine
      while           -- while mit Argumenten:
        (return (zeile /= [])) -- Bedingungsarg.
        (do putStrLn zeile      -- Aktionsarg.
            zeile <- getLine
            return ())
```

- ▶ Die Auswertung von `goUntilEmpty` terminiert nicht (es sei denn, `[]` wird als erste Eingabe gewählt).
- ▶ `zeile` und `zeile` sind unterschiedliche **Einmal-Wertvereinbarungen** gleichen Namens.
- ▶ Test und Ausgabe erfolgen bei jedem Aufruf von `while` (in jeder ‘Schleife’) für den Wert von `zeile`, nie v. `zeile`.

# Lösung: Direkte Rekursion statt 'Iteration'

Direkt-rekursive Lösung (ohne den iterativartigen Ausdruck `while`):

```
goUntilEmpty' :: IO ()
goUntilEmpty'
  = do zeile <- getLine
      if (zeile == [])
      then return ()
      else (do putStrLn zeile
                goUntilEmpty')    -- Rekursion!
```

(siehe Simon Thompson. [The Craft of Functional Programming](#). Addison-Wesley/Pearson, 2. Auflage, 1999, S. 393.)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 15.5.5

## Subtiles, Randbemerkung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# (Subtile) Unterschiede

...in **Wertdarstellung** und **Resultattyp** zwischen **Ausgabe-** und **Nichtausgabeoperationen**:

```
Main>putStr ('a':('b':('c':[])))  
->> abc :: IO ()
```

```
Main>putChar (head ['a','b','c'])  
->> a :: IO ()
```

```
Main>print "abc"  
->> "abc" :: IO ()
```

```
Main>print 'a'  
->> 'a' :: IO ()
```

```
Main>('a':('b':('c':[])))  
->> "abc" :: [Char]
```

```
Main>head ['a','b','c']  
->> 'a' :: Char
```

```
Main>"abc"  
->> 'a':('b':('c')) :: [Char]
```

```
Main>'a'  
->> 'a' :: Char
```



# Kapitel 15.6

## Zusammenfassung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Haskell-Programme als E/A-Aktionen

**Einstiegspunkt** für die Auswertung (übersetzter) interaktiver Haskell-Programme ist (per Konvention) eine **eindeutig** bestimmte

- ▶ Definition mit Namen **main** vom Typ **(IO T)**, **T** Typ.
- ▶ Intuitiv: 'Haskell-Programm = E/A-Aktion'.

Beispiel:

```
main :: IO ()                -- E/A-Schale
main
  = do n <- getInt           -- E/A-Schale
      let ergebnis = meine_funktion n -- Fkt. Kern
      putStr ergebnis       -- E/A-Schale

meine_funktion :: Int -> String -- Fkt. Kern
meine_funktion n = ... meine_funktion' ...

meine_funktion' :: ...        -- Fkt. Kern
meine_funktion' ... = ...
...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Ein-/Ausgabebehandlung

...in **funktionaler** und **imperativer** Programmierung grundsätzlich unterschiedlich. Am augenfälligsten:

- ▶ **Imperativ**: Ein-/Ausgabe prinzipiell an jeder Programmstelle möglich.
- ▶ **Funktional**, hier in **Haskell**: Ein-/Ausgabe an bestimmten Programmstellen konzentriert (in meist wenigen global definierten Funktionen der **'E/A-Schale'**).

**Häufige** Beobachtung: Die vermeintliche Einschränkung erweist sich

- ▶ als **Stärke** bei der **Programmierung im Großen!**

# Ein-/Ausgabebehandlung in Haskell

Haskells Konzept zur Behandlung von Ein-/Ausgabe erlaubt Funktionen

- ▶ des **Berechnungskerns** (**rein** funktionales Verhalten, keine Seiteneffekte)
- ▶ der **Dialog-** und **Interaktionsschale** (**nicht rein** funktionales, sondern **seiteneffektbehaftetes** Verhalten)

zu unterscheiden (und konzeptuell zu trennen), kenntlich an den unterschiedlichen Typen, auf deren Werten sie operieren:

**Int**, **Real**, **Char**, ... vs. **IO Int**, **IO Real**, **IO Char**, ...  
mit **IO** vordefinierter **Typkonstruktor** (wie z.B. **[]**, **(,)**, **(→)**).

Mithilfe der Kompositionsoperationen **(>=>)** und **(>>)** und der davon abgeleiteten gleichwertigen **do**-Notation ('**syntaktischer Zucker**') läßt sich die **Abfolge** von

- ▶ Ein-/Ausgabeoperationen **präzise** festlegen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Strombasierte Ein-/Ausgabebehandlung (1)

Frühe **Haskell**-Versionen haben eine **strombasierte** Behandlung von **Ein-/Ausgabe** vorgesehen:

- ▶ Programme werden dabei als Funktionen auf **Strömen** angesehen:  $EA\_PROG :: STRING \rightarrow STRING$



Peter Pepper. *Funktionale Programmierung*.  
Springer-Verlag, 2003, S. 271.

...mit **Ein-/Ausgabeströmen** für Terminals, Dateisysteme, Drucker, etc.

# Strombasierte Ein-/Ausgabebehandlung (2)

...Vor- und Nachteile für Sprachen mit:

- ▶ **sofortiger** (engl. **eager**) Auswertung:
  - ▶ ein 'echtes' Strommodell **fehlt** (die Eingabe muss zum Programmstart vollständig vorliegen und konsumiert werden und deshalb endlich sein); Ein-/Ausgabe ist deshalb auf stapelartige (engl. batch-like) Verarbeitung beschränkt.
- ▶ **verzögerter** (engl. **lazy**) Auswertung:
  - ▶ Interaktion ist möglich; verzögerte Auswertung stellt sicher, dass Ein-/Ausgaben in 'richtiger' Abfolge erfolgen.
  - ▶ **Aber:** Ursächlicher und zeitlicher Zusammenhang zwischen Ein-/Ausgaben erscheint oft 'obskur'; besondere Synchronisationen sind nötig, dies zu beheben.
  - ▶ **Insgesamt:** Strombasierte Ein-/Ausgabe kommt an ihre Grenzen beim Übergang zu graphischen Benutzerschnittstellen und wahlfreiem Zugriff auf Dateien.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

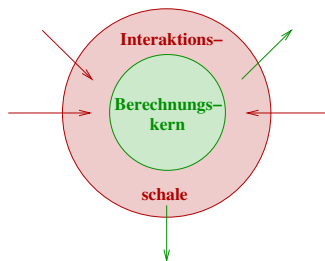
Kap. 12

Kap. 13

# Haskells heutige Lösung

...der konzeptuellen Trennung eines Haskell-Programms in

- ▶ einen rein funktionalen Berechnungskern
- ▶ eine Dialog- und Interaktionsschale



Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson, 2004, S. 89.

...wahrt das funktionale Paradigma und ist frei von den Problemen strombasierter Ein-/Ausgabebehandlung.

# Ausblick

...IO ist 1-stelliger Typkonstruktor und vordefinierte Instanz der Typ(konstruktor)klasse Monad:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
  m >> k = m >>= \_ -> k      -- Protoimpl. von (>>)
  fail   = error              -- Protoimpl. von fail
```

Mit IO für m erhalten wir für die Typsignaturen der Klassenfkt.:

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
(>>)   :: IO a -> IO b -> IO b
return :: a -> IO a
fail   :: a -> IO a      -- fail bislang unbenutzt
                        -- von uns.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17



# Ausblick (fgs.)

Die **Eigenschaften** bzw. **Anforderungen** von **Ein-/Ausgabe** an **funktionale Programmierung** und ihre **monadische** Behandlung in **Haskell** sind nicht **E/A**-spezifisch, sondern ein Beispiel von vielen, darunter:

- ▶ Seiteneffektbehaftete Programmierung
- ▶ Nichtdeterminismus
- ▶ Fehlerbehandlung
- ▶ Programmierung mit großen Datenstrukturen
- ▶ ...

Mehr dazu: LVA 185.A05 **Fortgeschrittene funktionale Programmierung**, jeweils im Sommersemester eines Studienjahrs.

# Kapitel 15.7

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 15 (1)

-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 10.1, The IO monad)
-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 17.5, Ein- und Ausgaben)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 7, Eingabe und Ausgabe)
-  Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 5, Ein-/Ausgabe; Kapitel 5.1, IO-Aktionen)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 15 (2)



Antonie J. T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 7.5, Input/Output in Functional Programming)



Andrew J. Gordon. *Functional Programming and Input/Output*. British Computer Society Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.



Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 16, Communicating with the Outside World)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 15 (3)

-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 10, Interactive programming)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 8, Input and output; Kapitel 9, More input and more output)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 21, Ein-/Ausgabe: Konzeptuelle Sicht; Kapitel 22, Ein-/Ausgabe: Die Programmierung)
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006. (Kapitel 18, Objekte und Ein-/Ausgabe)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 15 (4)

-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 7, I/O; Kapitel 9, I/O Case Study: A Library for Searching the Filesystem)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 18, Programming with actions)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 8, Playing the game: I/O in Haskell; Kapitel 18, Programming with monads)
-  Philip Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science 2:461-493, 1992.

Ein Mensch würde nie dazu kommen,  
etwas zu tun, wenn er stets warten würde,  
bis er es so gut kann, dass niemand mehr  
einen Fehler entdecken könnte.

John Henry Newman (1801-1890)  
engl. Kardinal

# Kapitel 16

## Fehlerbehandlung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 16.1

## Überblick, Orientierung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Typische Fehlersituationen und Sonderfälle

## Typische Fehlersituationen:

- ▶ Division durch 0.
- ▶ Zugriff auf das erste Element einer leeren Liste, `head []`.
- ▶ ...

## Typische Sonderfälle:

- ▶ Auseinanderfallen von **intendiertem** und **implementiertem Definitionsbereich** einer Funktion, z.B.
  - ▶  $! : \mathbb{N} \rightarrow \mathbb{N}$ : **Intendierter Definitionsbereich** ist  $\mathbb{N}$ .
  - ▶ `fac :: Integer -> Integer`: **Implementierter Definitionsbereich** ist  $\mathbb{Z}$  (abgesehen von Ressourcenbeschränkungen der Maschine).
- ▶ Umgang mit Argumentwerten außerhalb des **intendierten Definitionsbereichs**.
- ▶ ...

Jeder Fehler erscheint unheimlich dumm,  
wenn andre ihn begehen.

Georg Christoph Lichtenberg (1742-1799)  
dt. Physiker und Naturforscher

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Fehlersituationen und Sonderfälle

...bislang von uns *naiv* behandelt:

Typische Formulierungen aus den Aufgabenstellungen:

*...liefert die Funktion den vorher beschriebenen Wert als Resultat; anderenfalls...*

- ▶ *ist das Ergebnis*
  - ▶ die Zeichenreihe *"Ungültige Eingabe"*.
  - ▶ die leere Liste `[]`.
  - ▶ der Wert `0`.
  - ▶ ...
- ▶ *endet die Berechnung mit dem Aufruf*  
*error "Ungültige Eingabe"*.
- ▶ ...

Kleine Fehler in einem großen Werk sind die  
Brosamen, die man dem Neid hinwirft.

Claude Adrien Helvétius (1715-1771)  
franz. Philosoph

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17

# In diesem Kapitel

...beschreiben wir drei Möglichkeiten eines sukzessive **systematisch(er)en Umgangs** mit unerwarteten Programmsituationen und Fehlern:

- ▶ **Panikmodus** (Kap. 16.2)
- ▶ **Auffangwerte** (engl. **default values**) (Kap. 16.3)
  - ▶ Funktionsspezifisch
  - ▶ Aufrufspezifisch
- ▶ **Fehlertypen, Fehlerfunktionen** (Kap. 16.4)

Fremde Fehler haben wir vor Augen,  
unsere liegen uns im Rücken.

Seneca der Jüngere (um 4 v.Chr. - 65 n.Chr.)  
röm. Politiker, Philosoph und Schriftsteller

# Kapitel 16.2

## Panikmodus

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Panikmodus

## Ziel:

- ▶ Fehler und Fehlerursache melden, Berechnung stoppen.

## Hilfsmittel:

- ▶ Die polymorphe Funktion `error :: String -> a`.

## Wirkung:

### Der Aufruf

- ▶ `error "Funktion f: Ungültige Eingabe."`

### liefert die Meldung

- ▶ `Programmfehler: Funktion f: Ungültige Eingabe.`

und stoppt danach die Programmauswertung unwiderruflich.

# Anwendungsbeispiel

## Beispiel:

```
fac :: Integer -> Integer
```

```
fac n
```

```
  | n == 0    = 1
```

```
  | n > 0     = n * fac (n-1)
```

```
  | otherwise = error "Ungültige Eingabe."
```

```
fac 5    ->> 120
```

```
fac 0    ->> 1
```

```
fac (-5) ->> Programmfehler: Ungültige Eingabe.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Beurteilung des Panikmodus

## Positiv:

- ▶ Schnell und einfach umzusetzen.

## Negativ:

- ▶ Die Berechnung stoppt unwiderruflich.
- ▶ Jegliche Information über den Programmlauf ist verloren, auch sinnvolle.
- ▶ Für sicherheitskritische Systeme können die Folgen eines unbedingten Programmabbruchs fatal sein.

# Kapitel 16.3

## Auffangwerte

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Auffangwerte

## Ziel:

- ▶ Panikmodus vermeiden; Programmlauf nicht zur Gänze abbrechen, sondern Berechnung möglichst sinnvoll fortführen.

## Hilfsmittel: Verwendung von

- ▶ funktionsspezifischen (Variante 1)
- ▶ aufrufspezifischen (Variante 2)

Auffangwerten (engl. **default values**) im Fehlerfall.

# Variante 1: Funktionsspezifischer Auffangwert

## Auffangwertvariante 1:

- Im Fehlerfall wird ein **funktionsspezifischer** Wert als Resultat geliefert.

## Beispiel:

```
fac :: Integer -> Integer
fac n
  | n == 0      = 1
  | n > 0       = n * fac (n-1)
  | otherwise = -1
```

# Analyse des Beispiels

Im **Beispiel** der Funktion `fac` gilt:

- ▶ Negative Werte treten **nie als reguläres Resultat** einer Berechnung auf.
- ▶ Der **funktionsspezifische Auffangwert** `-1` erlaubt deshalb, negative Eingaben als fehlerhaft zu erkennen und zu melden, ohne den Programmlauf unwiderruflich abubrechen.
- ▶ Auch `n` selbst käme in diesem Beispiel sinnvoll als Auffangwert in Frage; die **aufrufspezifische Rückmeldung** beinhaltet so die ungültige Eingabe selbst, begünstigte dadurch die Fehlersuche und wäre daher sogar aussagekräftiger (Spezialfall von **Auffangwertvariante 2, aufrufspezifischer Auffangwert**)

Für **beide Auffangwertwahlen** in diesem Beispiel (`-1`, `n`) gilt:

- ▶ Die Fehlersituation ist für den Programmierer **transparent**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Beurteilung der Auffangwertvariante 1

## Positiv

- ▶ Panikmodus vermieden, Programmablauf nicht abgebrochen.

## Negativ

- ▶ Oft gibt es einen zwar naheliegenden und plausiblen funktionsspezifischen Auffangwert; jedoch kann dieser das Eintreten der Fehlersituation verschleiern und intransparent machen, wenn der Auffangwert auch als Resultat einer regulären Berechnung auftreten kann.
- ▶ Oft fehlt ein naheliegender und plausibler Wert als Auffangwert; die Wahl eines Auffangwerts ist in diesen Fällen willkürlich und unintuitiv.
- ▶ Oft fehlt ein funktionsspezifischer Auffangwert gänzlich; Auffangwertvariante 1 ist in diesen Fällen nicht anwendbar.

...dazu zwei Beispiele.

# Auffangwert vorhanden, aber verschleiernd

Beispiel:

```
rest :: [a] -> [a]
rest (_:xs) = xs
rest []      = []
```

Die Verwendung von `[]` als funktionsspezifischem Auffangwert

- ▶ liegt nahe und ist plausibel.

Allerdings:

- ▶ Das Auftreten der Fehlersituation wird verschleiert und bleibt für den Programmierer intransparent, da `[]` auch als reguläres Resultat einer Berechnung auftreten kann:

```
rest [42] ->> [] -- [] als reguläres Resultat:
                  -- Keine Fehlersituation!
rest []    ->> [] -- [] als irreguläres Resultat:
                  -- Fehlersituation eingetreten!
```

# Kein (naheliegender) Auffangwert vorhanden

Beispiel:

```
kopf :: [a] -> a  
kopf (u:_) = u  
kopf []    = ???
```

Ohne Kenntnis der **Instanz** von **a** ist

- ▶ ein **a**-Wert überhaupt nicht angebar: Auffangwert fehlt völlig.

Auch mit Kenntnis der **Instanz** von **a**, z.B., **kopf :: [Int] -> Int**, bietet sich

- ▶ kein **Int**-Wert als Auffangwert an: Naheliegender, plausibler Auffangwert fehlt.

...deshalb Übergang zu **Auffangwertvariante 2** mit **aufrufspezifischen Auffangwerten** als **mitübergebenem Argument**.

# Variante 2: Aufrufspezifische Auffangwerte (1)

## Auffangwertvariante 2:

- Im Fehlerfall wird ein **aufrufspezifischer** Auffangwert als Resultat geliefert. Dazu wird die Signatur erweitert und der jeweils gewünschte Auffangwert als Argument mitgeführt.

**Beispiel:** Ersetze **head** durch **head'** mit Typisierung

**kopf'** :: **a** -> [**a**] -> **a**

**kopf'** **\_** (**u** : **\_**) = **u**

**kopf'** **x** [] = **x**

...und **aufrufspezifischem** Auffangargument **x**.

## Variante 2: Aufrufspezifische Auffangwerte (2)

Generelle Vorgehensweise:

- Ergänze die fehlerbehandlungsfreie Implementierung einer (hier einstellig angenommenen) Funktion `f`:

```
f :: a -> b
```

```
f u = ...
```

um eine fehlerbehandelnde Variante `f'` dieser Funktion:

```
f' :: b -> a -> b
```

```
f' x u
```

```
  | fehlerFall = x
```

```
  | otherwise  = f u
```

wobei `fehlerFall` die Fehlersituation charakterisiert.

**Bemerkung:** Im Beispiel der Funktion `kopf'` konnte die Abstützung auf `kopf` gemäß der generellen Vorgehensweise umgangen werden.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Beurteilung der Auffangwertvariante 2 (1)

Positiv:

- ▶ **Panikmodus vermieden**, Programmlauf nicht abgebrochen.
- ▶ **Generalität**, stets anwendbar.
- ▶ **Flexibilität**, aufrufspezifische Auffangwerte ermöglichen variierende Fehlerwerte und Fehlerbehandlung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Beurteilung der Auffangwertvariante 2 (2)

## Negativ:

- ▶ **Transparente Fehlerbehandlung** ist nicht gewährleistet, wenn aufrufspezifische Auffangwerte auch reguläres Resultat einer Berechnung sein können, z.B.:  
kopf' 'F' "Fehler" ->> 'F' -- 'F' als reguläres Ergebnis  
kopf' 'F' "" ->> 'F' -- 'F' als irreguläres Ergebnis
- ▶ In diesen Fällen **Gefahr ausbleibender Fehlerwahrnehmung** mit (möglicherweise **fatalen**) Folgen durch
  - ▶ Vortäuschen eines regulären und korrekten Berechnungsablaufs und eines regulären und korrekten Ergebnisses!  
(Typischer Fall eines "sich ein 'x' für ein 'u' vormachen zu lassen!")

Der schlimmste aller Fehler ist,  
sich keines solchen bewusst zu sein.

Thomas Carlyle (1795-1881)  
schott. Essayist und Historiker

# Kapitel 16.4

## Fehlertypen, Fehlerfunktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Ziel

...systematisches

- ▶ Erkennen, Anzeigen und Behandeln von Fehlersituationen.

Hilfsmittel

- ▶ Dezidierte Fehlertypen, Fehlerwerte und Fehlerfunktionen statt schlichter Auffangwerte.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Zentral: Anzeigbarkeit von Fehlern

...Fehler werden **anzeigbar** durch Übergang von Typ **a** zum (Fehler-) Datentyp **Maybe a**

```
data Maybe a = Just a
              | Nothing
              deriving (Eq, Ord, Read, Show)
```

...die Werte des Typs **a** in der Form **Just a** mit dem Zusatzwert **Nothing** als explizitem **Fehlerwert**.

Verwendungsbeispiel:

```
div' :: Int -> Int -> Maybe Int
```

```
div' n m
```

```
  | m /= 0 = Just (div n m)
```

```
  | m == 0 = Nothing
```

```
div' 13 5 ->> Just 2      -- Division geklappt
```

```
div' 13 0 ->> Nothing    -- Division gescheitert
```

# Systematisierung der Verwendungsbeispielidee

...am Beispiel einer einstellig angenommenen fehlerbehandlungsfreien Funktion `f`:

► Ergänze `f`:

```
f :: a -> b
```

```
f u = ...
```

um die fehlererkennende und -anzeigende Variante `f'`:

```
f' :: a -> Maybe b
```

```
f' u
```

```
  | fehlerFall = Nothing
```

```
  | otherwise  = Just (f u)
```

wobei `fehlerFall` die Fehlersituation charakterisiert.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Angewendet auf unser Eingangsbeispiel

...ergänze die (vordef.) nichtfehlerbehandelnde Funktion `div`:

```
div :: Int -> Int -> Int
div n m = ... -- Details implementierungsintern

div 13 5 ->> 2
div 13 0 ->> Programmabbruch mit Laufzeitfehler
```

um die fehlererkennende und -meldende Variante `div'`:

```
div' :: Int -> Int -> Maybe Int
div' n m
  | m == 0    = Nothing
  | otherwise = Just (div n m)

div' 13 5 ->> Just 2      -- Division geklappt
div' 13 0 ->> Nothing    -- Division gescheitert
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Analyse, Diskussion

...anders als die Funktion `div`, deren Auswertung im Fehlerfall (d.h. Division durch 0)

- ▶ gemäß des `Panikmodus`

vom `Laufzeitsystem` abgebrochen wird, ist die Funktion `div'` in der Lage, einen Fehler ohne Auswertungsabbruch

- ▶ zu `erkennen`: `m == 0`
- ▶ in Gestalt des Resultats `anzuzeigen`: `Nothing`

Noch offen:

- ▶ Was machen wir im Fehlerfall mit dem Resultat `Nothing`?



# Generalisierung der bisherigen Idee

Ziel:

Erkennen, weiterreichen, fangen und behandeln von Fehlern mithilfe der Funktionen:

- ▶ `map_Maybe`: Erkennen und weiterreichen von Fehlern.
- ▶ `maybe`: Fangen und behandeln von Fehlern.

...erlauben im Zusammenspiel das Erkennen, Weiterreichen, Fangen u. schließliche Behandeln von Fehlern zu organisieren.

Beachte: `map_Maybe` ist verschieden von der im Standard-Prälium definierten namensähnlichen Funktion `mapMaybe` mit Signatur:  
`mapMaybe :: (a -> Maybe b) -> [a] -> [b]`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Die Funktion `map_Maybe`

...die Funktion `map_Maybe`:

```
map_Maybe :: (a -> b) -> Maybe a -> Maybe b
map_Maybe f Nothing   = Nothing      -- Durchreichen von
                                     -- Fehlern
map_Maybe f (Just u) = Just (f u)    -- Rechnen mit f im
                                     -- Normalfall
```

Curryfizierte und uncurryfizierte Sicht auf `map_Maybe`:

- ▶ **Curryfiziert:** `map_Maybe` bildet eine (nicht fehlerbehandelnde) Funktion vom Typ `(a -> b)` auf eine Funktion vom Typ `(Maybe a -> Maybe b)` ab (entspricht einem 'Typ-Lifting').
- ▶ **Uncurryfiziert:** `map_Maybe` bildet einen `(Maybe a)`-Wert auf einen `(Maybe b)`-Wert ab mithilfe einer (nicht fehlerbehandelnden) Funktion vom Typ `(a -> b)`.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Die Funktion maybe

...die Funktion maybe:

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe x f Nothing  = x      -- Aufrufspez. Fehlerwert
maybe x f (Just u) = f u    -- Rechnen mit f im Normalfall
```

Curryfizierte und uncurryfizierte Sicht auf map\_Maybe:

- ▶ **Curryfiziert:** Gegeben einen **b**-Wert bildet **maybe** eine Funktion vom Typ  $(a \rightarrow b)$  auf eine Funktion vom Typ  $(\text{Maybe } a \rightarrow b)$  ab (entspricht einem 'Typ-Lifting').
- ▶ **Uncurryfiziert:** **maybe** bildet einen  $(\text{Maybe } a)$ -Wert auf einen **b**-Wert ab mithilfe einer (nicht fehlerbehandelnden) Funktion vom Typ  $(a \rightarrow b)$  und eines aufrufspezifischen Fehlerarguments vom Typ **b** (entspricht der **Auffangwert-variante 2**).

# Im Zusammenspiel

...erlauben `map_Maybe` und `maybe` Fehlerwerte

- ▶ `weiterzureichen`, die Fähigkeit von `map_Maybe`:

```
map_Maybe f Nothing = Nothing -- Der Fehlerwert
                                -- Nothing wird
                                -- von map_Maybe
                                -- durchgereicht.
```

- ▶ zu `fangen` und (im Sinn von `Auffangwertvariante 2`) zu `behandeln`, die Fähigkeit von `maybe`:

```
maybe x f Nothing = x -- Der aufrufspezifische
                        -- Auffangwert x wird als
                        -- Resultat geliefert
                        -- (Auffangwertvariante 2)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Anwendungsbeispiel

...zum Zusammenspiel von `map_Maybe` und `maybe`:

- **Fehlerfall:** Der Fehler wird von `div'` erkannt und angezeigt, von `map_Maybe` weitergereicht und schließlich von `maybe` gefangen und behandelt.

```
maybe 9999 (+1) (map_Maybe (*3) (div' 9 0))  
->> maybe 9999 (+1) (map_Maybe (*3) Nothing)  
->> maybe 9999 (+1) Nothing  
->> 9999
```

- **Fehlerfreier Fall:** Alles läuft 'normal' ab.

```
maybe 9999 (+15) (map_Maybe (*3) (div' 9 1))  
->> maybe 9999 (+15) (map_Maybe (*3) (Just 9))  
->> maybe 9999 (+15) (Just 27)  
->> (+15) 27  
->> 27 + 15  
->> 42
```

# Bewertung d. Fehlerbehandlung mittels Maybe

## Positiv:

- ▶ Fehler können erkannt, angezeigt, weitergereicht und schließlich gefangen und (im Sinn von Auffangwertvariante 2) behandelt werden.

## Negativ:

- ▶ Geänderte Funktionalität: `Maybe b` statt `b`.

## Pragmatischer Zusatzvorteil:

- ▶ Systementwicklung ist ohne explizite Fehlerbehandlung möglich (z.B. mit nichtfehlerbehandelnden Funktionen wie `div`).
- ▶ Fehlerbehandlung kann nach Abschluss durch Ergänzung der fehlerbehandelnden Funktionsvarianten (wie z.B. der Funktion `div'`) zusammen mit den Funktionen `map_Maybe` und `maybe` umgesetzt werden.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17

Die schlimmsten Fehler werden gemacht  
in der Absicht, einen begangenen Fehler  
wieder gut zu machen.

Jean Paul (1763-1825)  
dt. Schriftsteller

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 16.5

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11




Teil V

Kap. 12

Kap. 13



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 16

-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 19, Error Handling)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 14.4, Case study: program errors)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 14.4, Modelling program errors)

Den größten Fehler, den man im Leben machen kann,  
ist, immer Angst zu haben, einen Fehler zu machen.

Dietrich Bonhoeffer (1906-1945)  
dt. luther. Theologe, NS-Widerständler

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

Ein Genie macht keine Fehler. Seine Irrtümer  
sind Tore zu neuen Entdeckungen.

James Joyce (1882-1941)  
irisch. Schriftsteller

Irrtümer haben ihren Wert;  
jedoch nur hie und da.  
Nicht jeder, der nach Indien fährt,  
entdeckt Amerika.

Erich Kästner (1899-1974)  
dt. Schriftsteller

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 17

## Module

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 17.1

## Überblick, Orientierung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Modularisierung

...wichtiges programmiersprachliches Hilfsmittel zur Dekomposition und Strukturierung von Programm(system)en für die Unterstützung der

- ▶ Programmierung im Großen.

In diesem Kapitel

- ▶ Ziele und Kennzeichen guter Modularisierung (Kap. 17.2)
- ▶ Haskell's Modulkonzept (Kap. 17.3)
- ▶ Spezielle Anwendung: Abstrakte Datentypen (Kap. 17.4)

Ich denke gern in großen Dimensionen.  
Wenn man schon denkt,  
kann man es ja auch gleich ordentlich tun.

Donald Trump (\* 1946)  
amerik. Unternehmer  
45. Präsident der USA

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 17.2

## Ziele und Richtlinien guter Modularisierung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Modularisierung

## Intuitiv:

- ▶ Zerlegung großer Programm(system)e in kleinere Einheiten, genannt **Module**.

## Ziel:

- ▶ Sinnvolle, über- und durchschaubare Organisation des Gesamtsystems.

# Modularisierungsgewinne

## Vorteile:

- ▶ **Arbeitsphysiologisch**: Unterstützung arbeitsteiliger Programmierung.
- ▶ **Softwaretechnisch**: Unterstützung der Wiederbenutzung von Programmen und Programmteilen.
- ▶ **Implementierungstechnisch**: Unterstützung getrennter Übersetzung (engl. separate compilation).

## Insgesamt:

- ▶ Höhere Effizienz der **Softwareerstellung** bei gleichzeitiger Qualitätssteigerung (Verlässlichkeit) und **Kostenreduktion**.



# Modularisierungsanforderungen

...zur Erreichung vorgenannter Ziele.

Unterstützung des Geheimnisprinzips durch Trennung von

- ▶ Schnittstelle (Import/Export)
  - ▶ Wie interagiert das Modul mit seiner Umgebung?
  - ▶ Welche Funktionalität stellt es zur Verfügung (Export)?
  - ▶ Welche Funktionalität benötigt es (Import)?
- ▶ Implementierung (Daten/Funktionen)
  - ▶ Wie sind die Datenstrukturen implementiert?
  - ▶ Wie ist die Funktionalität auf den Datenstrukturen realisiert?

# Regeln 'guter' Modularisierung

**Lokale Sicht:** Jedes Modul soll

- ▶ einen klar definierten, unabhängig von anderen Modulen verständlichen Zweck besitzen.
- ▶ nur einer Abstraktion entsprechen.
- ▶ einfach zu testen sein.

**Globale Sicht:** In modular entworfenen Programmen sollen

- ▶ Auswirkungen von Designentscheidungen (z.B. Einfachheit vs. Effizienz einer Implementierung)
- ▶ Abhängigkeiten von anderen Programmen oder Hardware

...auf (möglichst) wenige Module beschränkt sein.

# Modularisierungseigenschaften

...zentral:

- ▶ **Intramodular: Kohäsion**
  - ▶ beschäftigt sich mit Art und Typ der in einem Modul zusammengefassten Funktionen.
- ▶ **Intermodular: Koppelung**
  - ▶ beschäftigt sich mit dem Import-/Export- und Datenaustauschverhalten von Modulen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Intramodular: Kohäsion

...anzustreben:

- ▶ **Funktionale Kohäsion** (Funktionen gleicher Funktionalität sind in einem Modul zusammengefasst, z.B. Sortiervverfahren, Ein-/Ausgabefunktionen, etc.).
- ▶ **Datenkohäsion** (Auf gleichen Datenstrukturen arbeitende Funktionen sind in einem Modul zusammengefasst, z.B. Funktionen auf trigonometrischen Daten, auf Wasserstandsdaten, etc.).

...zu vermeiden:

- ▶ **Logische Kohäsion** (Funktionen vergleichbarer Funktionalität mit unterschiedlicher Implementierung sind in einem Modul zusammengefasst, z.B. verschiedene Benutzerschnittstellen eines Systems).
- ▶ **Zufällige Kohäsion** (Funktionen sind sachlich unbegründet in einem Modul zusammengefasst, zufällig eben).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Intermodular: Koppelung

...anzustreben:

- ▶ **Schwache funktionale Koppelung**, d.h. wenige, wohlbe-gründete funktionale Beziehungen und Abhängigkeiten zwischen verschiedenen Modulen.
- ▶ **Feste Datenkoppelung**, d.h. durch Wertübergabe (Funk-tionen unterschiedlicher Module kommunizieren nur durch explizite Übergabe von Werten, d.h. Ergebnisse einer Funktion werden Argument einer anderen Funktion.).

..zu vermeiden:

- ▶ **Starke funktionale Koppelung**.
- ▶ **Lose Datenkoppelung**, d.h. durch andere Mechanismen als Wertübergabe, z.B. Kommunikation über Dateien.

**Bemerkung:** Datenkoppelung durch Wertübergabe ist in funk-tionalen Sprachen *per se* als Grundform gegeben.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Ziel und Kennzeichen 'guter' Modularisierung

## Starke funktionale und Datenkohäsion

- ▶ enger inhaltlicher Zusammenhang der Definitionen eines Moduls.

## Schwache funktionale und lose Datenkoppelung

- ▶ wenige Abhängigkeiten zwischen verschiedenen Modulen, insbesondere keine direkten oder indirekten zirkulären Abhängigkeiten.

Für eine weitergehende und vertiefende Diskussion siehe:

- ▶ Manuel Chakravarty, Gabriele Keller. [Einführung in die Programmierung mit Haskell](#), Pearson Studium, 2004, Kapitel 10.

# Kapitel 17.3

## Haskells Modulkonzept

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Schematischer Aufbau eines Haskellmoduls

```
module M where                -- Moduldefinition

data D_1 ... = ...           -- Datentypdefinitionen
...

data D_n ... = ...

newtype N_1 ... = ...        -- Neue Typendefinitionen
...

newtype N_m ... = ...

type T_1 ... = ...           -- Typsynonymdefinitionen
...

type T_p ... = ...

class C_1 ...                -- Typklassendefinitionen
...

class C_q ...

f_1 :: ...                   -- Funktionsdefinitionen
f_1 ... = ...
...

f_r :: ...
f_r ... = ...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Haskells Modulkonzept

...unterstützt:

- ▶ Export

- ▶ Selektiv/nicht selektiv
- ▶ Händischer Reexport
- ▶ **Nicht unterstützt:** Automatischer Reexport

- ▶ Import

- ▶ Selektiv/nicht selektiv
- ▶ Qualifiziert
- ▶ Mit Umbenennung

von Datentypen, Typsynonymen, Typklassen und Funktionen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 17.3.1

## Import

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Import: Nicht selektiv

## Allgemeines Muster:

```
module M1 where
...
```

```
module M2 where
import M1 -- Alle im Modul M1 (global sichtbaren) Be-
           -- zeichner, Definitionen werden importiert
           -- und können im Modul M2 benutzt werden.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Import: Selektiv

Allgemeines Muster, zwei Varianten:

```
module M1 where...
```

```
module M2 where
```

```
import M1 (D_1 (...), D_2, T_1, C_1 (...), C_2, f_5)
-- Ausschließlich D_1 (einschließlich von M1 exportierter
-- tierter Konstruktoren), D_2 (ohne Konstruktoren),
-- T_1, C_1 (...) (einschließlich von M1 exportierter
-- Funktionen), C_2 (ohne Funktionen), f_5 werden aus
-- M1 importiert und können in M2 benutzt werden,
-- d.h., importiere nur, was explizit genannt ist!
```

```
module M3 where
```

```
import M1 hiding (D_1, T_2, f_1)
-- Alle (sichtbaren) Bezeichner, Definitionen aus M1
-- mit Ausnahme der explizit genannten werden importiert
-- und können in M3 benutzt werden, d.h., importiere
-- alle, was nicht explizit ausgeschlossen wird!
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Anwendungsbeispiel

...`'verstecken'` der im `Standard-Präludium` vordefinierten Funktionen

▶ `reverse`, `tail` und `zip`

durch Einfügen von

▶ `import Prelude hiding (reverse,tail,zip)`

am Anfang des Haskell-Skripts im Anschluss an die (so vorhandene) Modul-Anweisung (siehe [Kapitel 1.3.1](#)).

# Kapitel 17.3.2

## Export

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Export: Nicht selektiv

## Allgemeines Muster:

```
module M1 where      -- Alle im Modul M1 eingeführten
data D_1 ... = ...   -- global sichtbaren Bezeichner,
...                  -- Definitionen werden exportiert
data D_n ... = ...   -- und können von anderen Modulen
newtype N_1 ... = ... -- importiert werden.
...
newtype N_m ... = ...
type T_1 = ...
...
type T_p = ...
class C_1 ...
...
class C_q ...        -- Beachte: module M1 where...
f_1 :: ...           -- ist bedeutungsgleich zu
f_1 ... = ...        -- module M1 (module M1) where...
...
f_r :: ...
f_r ... = ...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Export: Selektiv

## Allgemeines Muster:

```
module M1 (D_1 (...), D_2, D_3 (Dc_1,...,Dc_k), C_1 (...),
          C_2, C_3 (cf_1,...,cf_l), T_1, f_2, f_5) where
data D_1 ... = ...
...
data D_n ... = ...
newtype N_1...=...
...
newtype N_m...=...
type T_1 ... = ...
...
type T_p ... = ...
class C_1 ...
...
class C_q ...
f_1 :: ...
f_1 ... = ....
...
f_r :: ...
f_r ... = ...
```

-- Nur die explizit genannten Bezeich-  
ner, Definitionen werden aus M1 ex-  
portiert und können von anderen Mo-  
dulen importiert werden. Dabei gilt:  
D\_1 wird einschließlich seiner Kon-  
strukturen exportiert; D\_2 ohne; D\_3  
mit den explizit genannten. Analog  
für die Klassen C\_i.

-- Selektiver Export unterstützt  
-- das Geheimnisprinzip!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Kapitel 17.3.3

## Reexport

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Reexport: Nicht automatisch, nur händisch

## Veranschaulichung:

```
module M1 where...
```

```
module M2 where
```

```
import M1 -- Nicht selektiver Import aus M1, d.h. alle  
...      -- in M1 (global sichtbaren) Bezeichner, De-  
f_2M2    -- finitionen werden von M2 importiert und  
...      -- können in M2 benutzt werden.
```

```
module M3 where
```

```
import M2 -- Nicht selektiver Import aus M2, d.h. alle  
...      -- in M2 (global sichtbaren) Bezeichner, De-  
-- finitionen werden von M3 importiert und  
-- können in M3 benutzt werden, nicht jedoch  
-- die von M2 aus M1 importierten Namen,  
-- d.h. kein automatischer Reexport!
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Reexport: Händisch

**Abhilfe:** Händischer Reexport, zwei Varianten:

```
module M2 (module M1,f_2M2) where -- Nicht selektiver Reex-
import M1                        -- port von M1 aus M2:
...                             -- M2 reexportiert jeden
f_2M2                           -- aus M1 importierten
                                -- Namen, sowie das M2-
                                -- lokale f_2M_2 aus M2.
...

module M2 (D_1 (...), D_2, D_3 (Dc_1,Dc_2), C_1 (...), C_2,
          C_3 (cf_1,cf_2,cf_3), f_1,f_2M2) where
import M1 -- Selektiver Reexport von M1 aus M2: M2 reex-
...      -- portiert von den aus M1 importierten Namen
f_2M2    -- ausschließlich D_1 (einschließlich Konstruk-
...      -- toren), D_2 (ohne Konstruktoren), D_3 (mit
          -- angegebenen Konstruktoren); analog für die
          -- Klassen C_1, C_2 und C_3, sowie f_1 und das
          -- M2-lokale f_2M_2.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 17.3.4

## Namenskonflikte, Umbenennungen, Konventionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Namenskonflikte, Umbenennungen

## Namenskonflikte

- ▶ können durch **qualifizierten Import** aufgelöst werden:

```
import qualified M1
```

Verwendung: **M1.f** zur Bezeichnung der aus **M1** importierten Funktion **f**; **f** zur Bezeichnung der im importierenden Modul lokal definierten Funktion **f**.

## Umbenennen importierter Module und Bezeichner

- ▶ durch Einführen lokaler Namen im importierenden Modul

- ▶ für **Modulnamen**:

```
import qualified M1 as MyLocalNameForM1
```

...**MyLocalNameForM1** wird im importierenden Modul anstelle von **M1** verwendet.

- ▶ für **ausgewählte Bezeichner**:

```
import M1 (f1,f2)
```

```
renaming (f1 to fac, f2 to fib)
```

# Konventionen, gute Praxis

## Konventionen

- ▶ Pro Datei **ein** Modul.
- ▶ Modul- und Dateiname stimmen überein (abgesehen von der Endung **.hs** bzw. **.lhs** im Dateinamen).
- ▶ Alle Deklarationen beginnen in derselben Spalte wie das Schlüsselwort **module**.

## Gute Praxis

- ▶ Module unterstützen **eine (!)** klar abgegrenzte Aufgabenstellung (vollständig) und sind in diesem Sinne in sich abgeschlossen; ansonsten Teilen (Teilungskriterium).
- ▶ Module sind **'kurz'** (d.h. 'so kurz wie möglich, so lang wie nötig'; Faustregel: zwei bis drei Bildschirmseiten).

# Haskell-Programme

...sind **Modulsysteme**.

Soll ein Haskell-Programm **übersetzt** (statt **interpretiert**) werden, muss dessen Modulsystem ein **Hauptmodul** namens

► **Main**

mit einer Funktion namens

► **main :: IO  $\tau$**  für einen Typ  $\tau$

enthalten, mit deren Auswertung die Ausführung des übersetzten Programms beginnt (wobei das Ergebnis vom Typ  $\tau$  unbeachtet bleibt).

**Bemerkung:** Die **module**-Deklaration darf in einem Haskell-Skript fehlen; implizit wird in diesem Fall die **module**-Deklaration

```
module Main (main) where
```

ergänzt.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 17.4

## Abstrakte Datentypen als Modul-Anwendung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Abstrakte Datentypen

...als spezielle Anwendung des **Modulkonzepts** in **Haskell**.

Mit **abstrakten Datentypen (ADTs)** verfolgtes Ziel:

- ▶ Kapselung von Daten, Realisierung des **Geheimnisprinzips** auf Datenebene (engl. **information hiding**).

**Implementierungstechnischer Schlüssel:**

- ▶ Haskell's Modulkonzept, speziell der **selektive Export**, bei dem Konstruktoren algebraischer Datentypen verborgen bleiben.

# Grundlegende Idee von ADT-Definitionen (1)

...implizite Festlegung eines Datentyps in zwei Teilen:

- ▶ A) **Schnittstellenfestlegung**: Angabe der auf den Werten des Datentyps zur Verfügung stehenden Operationen in Form ihrer syntaktischen Signaturen.
- ▶ B) **Verhaltensfestlegung**: Festlegung der Bedeutung der Operationen durch Angabe ihres Zusammenspiels in Form von **Axiomen** (oder sog. **Gesetzen**), die von jeder (!) Implementierung dieser Operationen einzuhalten sind.

**Beachte:** Die Darstellung der Werte des abstrakten Datentyps wird ausdrücklich nicht festgelegt; sie bleibt verborgen und deshalb für die Implementierung als Freiheitsgrad offen!

# Grundlegende Idee von ADT-Definitionen (2)

## Herausforderung:

- ▶ Die Gesetze so zu wählen, dass das Verhalten der Operationen präzise und eindeutig festgelegt ist; also so, dass weder eine **Überspezifikation** (keine widerspruchsfreie Implementierung möglich) noch eine **Unterspezifikation** (mehrere in sich widerspruchsfreie, aber sich widersprechende Implementierungen möglich) vorliegt.

## Pragmatischer Gewinn:

- ▶ Die Trennung von Schnittstellen- und Verhaltensfestlegung erlaubt die Implementierung zu verstecken (**Geheimnisprinzip!**) und nach Zweckmäßigkeit und Anforderungen (z.B. Einfachheit, Performanz) auszuwählen und auszutauschen.

# Beispiel: FIFO-Warteschlange als ADT

...in Pseudo-Code (kein Haskell):

A: Festlegung der Schnittstelle durch Signaturangabe:

```
NEW:                                -> Queue
ENQUEUE: Queue  $\times$  Item -> Queue
FRONT:   Queue                    -> Item
DEQUEUE: Queue                    -> Queue
IS_EMPTY: Queue                   -> Boolean
```

B: Festlegung der einzuhaltenden Axiome/Gesetze:

```
b1) IS_EMPTY(NEW)                = true
b2) IS_EMPTY(ENQUEUE(q,i))       = false
b3) FRONT(NEW)                   = error
b4) FRONT(ENQUEUE(q,i))          = if IS_EMPTY(q) then i
                                   else FRONT(q)
b5) DEQUEUE(NEW)                 = error
b6) DEQUEUE(ENQUEUE(q,i))        =
    if IS_EMPTY(q) then NEW
    else ENQUEUE(DEQUEUE(q),i)
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# FIFO-Warteschlange: Haskell-Realisierung (1)

```
module Queue
```

```
{- Kommunikation von Teil A: Schnittstellenspezifikation -}
```

```
(Queue,      -- Name des Datentyps (Geheimnisprinzip, kein  
             -- Konstruktorexport!)
```

```
new,        -- new :: Queue a
```

```
enqueue,    -- enqueue :: Queue a -> a -> Queue a
```

```
front,      -- front :: Queue a -> a
```

```
dequeue,    -- dequeue :: Queue a -> Queue a
```

```
is_empty,   -- is_empty :: Queue a -> Bool
```

```
{- Kommunikation von Teil B: Axiome/Gesetze
```

```
  b1) is_empty(new)           = True
```

```
  b2) is_empty(enqueue(q,i)) = False
```

```
  b3) front(new)              = error "Niemand wartet!"
```

```
  b4) front(enqueue(q,i))     = if is_empty(q) then i  
                                else front(q)
```

```
  b5) dequeue(new)            = error "Niemand wartet!"
```

```
  b6) dequeue(enqueue(q,i))   = if is_empty(q) then new  
                                else enqueue(dequeue(q),i) -}
```

```
) where...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# FIFO-Warteschlange: Haskell-Realisierung (2)

```
{- Impl. von Teil B: Als algebraischer Datentyp, B1 -}  
  
data Queue a = Qu [a]  
  
new :: Queue a  
new = Qu []  
  
enqueue :: Queue a -> a -> Queue a  
enqueue (Qu xs) x = Qu (xs ++ [x])  
  
front :: Queue a -> a  
front q@(Qu xs)      -- Hier praktisch: Das als-Muster  
  | not (is_empty q) = head xs  
  | otherwise        = error "Schlange leer; niemand wartet!"  
  
dequeue :: Queue a -> Queue a  
dequeue q@(Qu xs)    -- Hier praktisch: Das als-Muster  
  | not (is_empty q) = Qu (tail xs)  
  | otherwise        = error "Schlange leer; niemand wartet!"  
  
is_empty :: Queue a -> Bool  
is_empty (Qu []) = True  
is_empty _       = False
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# FIFO-Warteschlange: Haskell-Realisierung (3)

```
{- Implementierung von Teil B: Als neuer Typ, B2 -}

newtype Queue a = Qu [a]

new :: Queue a
new = Qu []

enqueue :: Queue a -> a -> Queue a
enqueue (Qu xs) x = Qu (xs ++ [x])

front :: Queue a -> a
front q@(Qu xs)      -- Hier praktisch: Das als-Muster
  | not (is_empty q) = head xs
  | otherwise        = error "Schlange leer; niemand wartet!"

dequeue :: Queue a -> Queue a
dequeue q@(Qu xs)    -- Hier praktisch: Das als-Muster
  | not (is_empty q) = Qu (tail xs)
  | otherwise        = error "Schlange leer; niemand wartet!"

is_empty :: Queue a -> Bool
is_empty (Qu []) = True
is_empty _       = False
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# FIFO-Warteschlange: Haskell-Realisierung (4)

{- Implementierung von Teil B: Als Typsynonym, B3 -}

```
type Queue a = [a]
```

```
new :: Queue a
```

```
new = []
```

```
enqueue :: Queue a -> a -> Queue a
```

```
enqueue q x = q ++ [x]
```

```
front :: Queue a -> a
```

```
front q
```

```
  | not (is_empty q) = head q
```

```
  | otherwise        = error "Schlange leer; niemand wartet!"
```

```
dequeue :: Queue a -> Queue a
```

```
dequeue q
```

```
  | not (is_empty q) = tail q
```

```
  | otherwise        = error "Schlange leer; niemand wartet!"
```

```
is_empty :: Queue a -> Bool
```

```
is_empty q = (q == [])
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Das als-Muster

...als nützliche **Musterspielart** (siehe **Kapitel 6.1.5**):

```
front q@(Qu xs)    -- Arg. als q oder als (Qu xs).  
dequeue q@(Qu xs) -- Arg. als q oder als Qu xs).
```

Das **als**-Muster (**q@(Qu xs)**) bietet mittels:

- ▶ **q**: Zugriff auf das Argument als Ganzes.
- ▶ **(Qu xs)**: Zugriff auf Teile des strukturierten Arguments.

# Abstrakte vs. algebraische Datentypen

## Abstrakte Datentypen

- ▶ werden durch ihr **Verhalten spezifiziert**, d.h. durch die auf ihren Werten definierten Funktionen/Operationen und deren Zusammenspiel.
- ▶ die tatsächliche **Darstellung** der **Werte** des Datentyps wird zum Definitionszeitpunkt nicht angegeben u. **bleibt offen**.
- ▶ Dem Anwender eines abstrakten Datentyps wird die Darstellung der Werte und die Implementierung der Funktionen darauf nie bekanntgegeben: **Geheimnisprinzip!**

## Konkrete Datentypen (in Haskell: Algebraische Datentypen)

- ▶ werden durch die exakte Angabe und Darstellung ihrer Werte spezifiziert, aus denen sie bestehen.
- ▶ auf ihnen gegebene **Funktionen/Operationen** werden zum Definitionszeitpunkt nicht angegeben und **bleiben offen**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# ADT-Vorteile

...das **Geheimnisprinzip**: Nur die Schnittstelle ist bekannt, die Implementierung bleibt verborgen.

Das gewährleistet folgende Vorteile der **ADT**-Verwendung:

- ▶ Schutz der Datenstruktur vor unkontrolliertem oder nicht beabsichtigtem/zugelassenem Zugriff.

**Beispiel:** Ein eigendefinierter Leerheitstest wie:

```
emptyQ == Qu []
```

fürhte in **Queue** importierenden Modulen zu einem Laufzeitfehler, da die Implementierung und somit der Konstruktor **Qu** dort nicht sichtbar sind.

- ▶ Einfache Austauschbarkeit der zugrundeliegenden Implementierung.
- ▶ Unterstützung arbeitsteiliger Programmierung.

# Zur ADT-Realisierung in Haskell

...ADTs sind keine erstrangigen Sprachelemente (engl. *first class citizens*) in Haskell:

- ▶ Haskell bietet kein dezidiertes Sprachkonstrukt zur Spezifikation von ADTs, das eine externe Offenlegung von Signaturen und Gesetzen bei intern bleibender Implementierung erlaubt.

ADTs können in Haskell unter Ausnutzung des

- ▶ *Modulkonzepts* realisiert werden.

Das erlaubt, die *Implementierung*

- ▶ intern und damit im Sinn des Geheimnisprinzips versteckt zu halten
- ▶ jedoch können die Funktionssignaturen und Gesetze dem ADT-Anwender nur umständlich und in unsicherer Weise in Form von Kommentaren kommuniziert werden.

# Wegweisende Arbeiten

...zu **abstrakten Datentypen**:

- ▶ John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM 20(6):396-404, 1977.
- ▶ John V. Guttag, James Jay Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica 10(1):27-52, 1978.
- ▶ John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 17.5

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10




Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 17 (1)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 8, Modularisierung und Schnittstellen)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 10, Modularisierung und Programmdekomposition)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 6, Modules)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 17 (2)



John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM 20(6):396-404, 1977.



John V. Guttag, James Jay Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica 10(1):27-52, 1978.



John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11




Teil V

Kap. 12

Kap. 13



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 17 (3)

-  David L. Parnas. *On the Criteria to be used on Decomposing Systems into Modules*. Communications of the ACM 15(12):1053-1058, 1972.
-  David L. Parnas, Paul C. Clements, David M. Weiss. *The Modular Structure of Complex Systems*. IEEE Transactions on Software Engineering 11(3):259-266, 1985.
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 5, Writing a Library: Working with JSON Data – The Anatomy of a Haskell Module, Generating a Haskell Program and Importing Modules)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10




Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 17 (4)

-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 14, Datenstrukturen und Modularisierung)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 15.1, Modules in Haskell; Kapitel 15.2, Modular design; Kapitel 16, Abstract data types)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 15.1, Modules in Haskell; Kapitel 15.2, Modular design; Kapitel 16, Abstract data types)

# Kapitel 18

## Programmierprinzipien

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 18.1

## Überblick, Orientierung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Überblick, Orientierung

## Funktionen höherer Ordnung (Kap. 18.2)

- ▶ ermöglichen **algorithmisches Vorgehen** zu verpacken.  
Illustrierendes Beispiel: **Teile und Herrsche**.

## Späte Auswertung (engl. lazy evaluation) (Kap. 18.3)

- ▶ ermöglicht **semantische Modularisierungsprinzipien**:  
**Generator/Selektor-, Generator/Filter-, Generator/Transformator-Prinzip** und Kombinationen davon.  
Illustrierendes Beispiel: **Programmieren mit Strömen** (unendliche Listen (engl. streams, lazy lists)).

## Reflektives Programmieren (Kap. 18.4)

- ▶ Stetes **Hinterfragen** und **Anpassen** des eigenen Vorgehens.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 18.2

## Teile und Herrsche

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Teile und Herrsche

...die zugrundeliegende **algorithmische Idee** des 'Teile und Herrsche'-Prinzips:

- ▶ Ist ein Problem **einfach genug**, löse es sofort.
- ▶ **Teile** anderenfalls das Problem **rekursiv** in kleinere Teilprobleme, bis alle **Teilprobleme einfach genug** sind, sofort gelöst werden zu können.
- ▶ Berechne die Lösung des ursprünglichen Problems aus den Lösungen der Teilprobleme.

...eine typische **von-oben-nach-unten**-Vorgehensweise (engl. **top-down**)!

Divide et impera.  
Teile und herrsche.

nach Ludwig XI von Frankreich (1423-1483)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

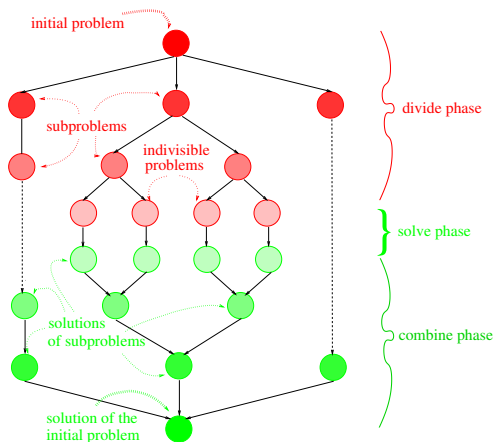
Teil V

Kap. 12

Kap. 13

# Veranschaulichung

Die Phasenabfolge des 'Teile und Herrsche'-Prinzips:



Fethi Rabhi, Guy Lapalme.

*Algorithms: A Functional Programming Approach.*

Addison-Wesley, 1999, Seite 156.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

1432/17



# Typische Anwendungsfelder

...und Anwendungen für Vorgehen mittels 'teile und herrsche':

- ▶ Sortierverfahren (Quicksort, Mergesort, etc.)
- ▶ Numerische Analyseverfahren
- ▶ Kryptographie
- ▶ Bildverarbeitung
- ▶ Binomialkoeffizientenberechnung
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Vorbereitung der funktionalen Umsetzung

## Gegeben:

- ▶ Ein **Problem** mit Probleminstanzen eines generischen Typs, beschrieben durch die Typvariable **pb**.

## Gesucht:

- ▶ Eine **Lösung** aus einer Menge von Lösungsinstanzen eines generischen Typs, beschrieben durch die Typvariable **lsg**.

## (Algorithmisches) Ziel:

Eine **Funktion höherer Ordnung** (oder **Funktional**) **teile\_und\_herrsche**, die geeignet parametrisiert für

- ▶ Probleminstanzen vom Typ **pb** gemäß des 'Teile und Herrsche'-Prinzips eine **Lösungsinstanz** vom Typ **lsg** berechnet.

# Parameter

... des Funktional `teile_und_herrsche`:

- ▶ `einfach_genug :: pb -> Bool`: ...liefert `True`, falls die Probleminstance einfach genug ist, um sofort gelöst werden zu können.
- ▶ `loese :: pb -> lsg`: ...liefert die Lösungsinstanz einer unmittelbar lösbaren Probleminstance.
- ▶ `teile :: pb -> [pb]`: ...teilt eine nicht unmittelbar lösbare Probleminstance in eine Liste von Teilprobleminstanzen auf.
- ▶ `herrsche :: pb -> [lsg] -> lsg`: ...liefert angewendet auf eine Ausgangsprobleminstance und eine Liste von Lösungen von Teilprobleminstanzen die Lösung der Ausgangsprobleminstance.

# Nützliche Typsynonyme

...auftretender Funktionstypen:

```
type Einfach_genug pb = pb -> Bool
```

```
type Loese pb lsg      = pb -> lsg
```

```
type Teile pb          = pb -> [pb]
```

```
type Herrsche pb lsg   = pb -> [lsg] -> lsg
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Das Funktional teile\_und\_herrsche

```
teile_und_herrsche :: (Einfach_genug pb) -> (Loese pb lsg)
                  -> (Teile pb) -> (Herrsche pb lsg) -> pb -> lsg
```

```
teile_und_herrsche einfach_genug loese teile herrsche
                  pb_instanz
```

```
= tuh pb_instanz
```

```
where
```

```
  tuh p
```

```
    | einfach_genug p = loese p
```

```
    | otherwise      = herrsche p (map tuh (teile p))
```

Löse rekursiv alle  
durch die Teilung  
entstehenden Probleme.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

1437/17

# Teile und Herrsche am Beispiel von Quicksort

```
quickSort :: Ord a => [a] -> [a]
quickSort liste
  = teile_und_herrsche einfach_genug loese teile
                        herrsche liste

where
  einfach_genug ls          = length ls <= 1
  loese             = id
  teile (l:ls)       = [[x | x <- ls, x <= l],
                        [x | x <- ls, x > l]]
  herrsche (l:_) [ls1,ls2] = ls1 ++ [l] ++ ls2
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Warnung

...nicht jedes Problem, das dem 'teile und herrsche'-Vorgehen in natürlicher Weise zugänglich ist, ist auch (in naiver Weise) dafür geeignet.

Betrachte dazu folgendes Beispiel:

```
fib :: Integer -> Integer
fib n = teile_und_herrsche einfach_genug loese
      teile herrsche n

where
  einfach_genug n      = (n == 0) || (n == 1)
  loese              = id
  teile n             = [n-2,n-1]
  herrsche _ [m1,m2] = m1 + m2
```

...besitzt exponentielles Laufzeitverhalten!

# Algorithmenmuster

Die Idee, ein generelles algorithmisches Vorgehen wie 'teile und herrsche' durch eine geeignete Funktion höherer Ordnung wiederverwendbar zu machen, lässt sich auch für andere algorithmische Verfahrensweisen umsetzen, darunter:

- ▶ Rücksetzsuche (engl. Backtracking Search)
- ▶ Prioritätsgesteuerte Suche
- ▶ Lokale Suche (engl. Greedy Search)
- ▶ Dynamische Programmierung

Wir sprechen hier auch von Algorithmenmustern (mehr dazu in der LVA 185.A05 'Fortgeschrittene funktionale Programmierung').



# Kapitel 18.3

## Stromprogrammierung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Ströme

...programmiersprachlicher **Jargon** für

- ▶ **unendliche Listen** (engl. **streams**, **lazy lists**).

**Ströme** ermöglichen im Zusammenspiel mit **später Auswertung** qualitativ neue

- ▶ semantikbasierte Modularisierungen
  - ▶ **Generator/Selektor**-Prinzip
  - ▶ **Generator/Filter**-Prinzip
  - ▶ **Generator/Transformator**-Prinzip

mit denen sich viele Probleme **elegant**, **knapp** und **effizient** lösen lassen.

# Ströme am Bsp. des Siebs des Eratosthenes (1)

...zur Berechnung des Stroms der Primzahlen:

1. Schreibe **alle** natürlichen Zahlen ab **2** hintereinander auf.
2. Die kleinste nicht gestrichene Zahl in dieser Folge ist eine **Primzahl**. Streiche alle Vielfachen dieser Zahl.
3. Wiederhole Schritt 2 mit der kleinsten jeweils noch nicht gestrichenen Zahl.

Nach Schritt 1:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19...

Nach Schritt 2 für Zahl 2:

2 3    5    7    9    11    13    15    17    19...

Nach Schritt 2 für Zahl 3:

2 3    5    7       11    13       17    19...

usw.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Ströme am Bsp. des Siebs des Eratosthenes (2)

```
primes :: [Integer]
primes = sieve [2..]

sieve :: [Integer] -> [Integer]
sieve (x:xs) = x : sieve [y | y <- xs, mod y x > 0]
```

Die (0-stellige) Funktion `primes` liefert

- den Strom der (unendlich vielen) Primzahlen.

Aufruf von `primes`:

```
primes ->> [2,3,5,7,11,13,17,19,23,29,31,37,41,...]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Ströme am Bsp. des Siebs des Eratosthenes (3)

Veranschaulichung der Stromberechnung durch händische Auswertung:

```
primes
->> sieve [2..]
->> 2 : sieve [y | y <- [3..], mod y 2 > 0]
->> 2 : sieve (3 : [y | y <- [4..], mod y 2 > 0])
->> 2 : 3 : sieve [z | z <- [ y | y <- [4..],
                                mod y 2 > 0],
                                mod z 3 > 0]

->> ...
->> 2 : 3 : sieve [z | z <- [5, 7, 9..],
                                mod z 3 > 0]

->> ...
->> 2 : 3 : sieve [5, 7, 11,...]
->> ...
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Semantische Modularisierungsprinzipien

...aus dem **Stromkonzept** erwachsen qualitativ neue, **semantische Modularisierungsprinzipien**.

Insbesondere:

- ▶ **Generator/Selektor-** (G/S-) Prinzip
- ▶ **Generator/Filter-** (G/F-) Prinzip
- ▶ **Generator/Transformator-** (G/T-) Prinzip

sowie Kombinationen davon wie das **G/T/S-** und **G/T/F-**Prinzip und weitere.

# G/S-Prinzip am Bsp. des Primzahlstroms (1)

Ein Generator (G):

- ▶ `genPrimes :: [Integer]`  
`genPrimes = primes`

Viele Selektoren (S):

- ▶ Nimm die ersten  $n$  Elemente einer Liste:  
`take :: Int -> [a] -> [a]`  
`take n ls = ...`
- ▶ Nimm das  $(n + 1)$ -te Element einer Liste,  $n \geq 0$ :  
`!! :: [a] -> Int -> a`  
`(!!) ls n = ...`
- ▶ Nimm alle ab dem  $(n + 1)$ -ten Element einer Liste:  
`drop :: Int -> [a] -> [a]`  
`drop n ls = ...`
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# G/S-Prinzip am Bsp. des Primzahlstroms (2)

Zusammenfügen der G/S-Module zum Gesamtprogramm:

► Anwendung des G/S-Prinzips:

Die ersten 5 Primzahlen:

```
take 5 genPrimes ->> [2,3,5,7,11]
```

► Anwendung des G/S-Prinzips:

Die 5-te Primzahl:

```
(!!) 4 genPrimes ->> genPrimes!!4 ->> 11
```

► Anwendung des G/S-Prinzips:

Die 6-te bis 10-te Primzahl:

```
take 5 (drop 5 genPrimes) ->> [13,17,19,23,29]
```



# G/F-Prinzip am Bsp. des Primzahlstroms (1)

Ein Generator (G):

- ▶ `gen_Primes :: [Integer]`  
`gen_Primes = primes`

Viele Filter (F):

- ▶ Alle Listenelemente größer als 1000:  
`filter (>1000) :: [Integer] -> [Integer]`  
`filter (>1000) ls = ...`
- ▶ Ist Zahl mit genau drei Einsen in der Dezimaldarstellung:  
`hat_drei_Einsen :: Integer -> Bool`  
`hat_drei_Einsen n = ...`
- ▶ Ist Zahl mit Palindromdezimaldarstellung:  
`ist_Palindrom :: Integer -> Bool`  
`ist_Palindrom n = ...`
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# G/F-Prinzip am Bsp. des Primzahlstroms (2)

Zusammenfügen der G/F-Module zum Gesamtprogramm:

► Anwendung des G/F-Prinzips:

Alle Primzahlen größer als 1000:

```
filter (>1000) genPrimes  
->> [1009,1013,1019,1021,1031,1033,1039,...]
```

► Anwendung des G/F-Prinzips:

Alle Primzahlen mit genau drei Einsen in der Dezimaldarstellung:

```
[ n | n <- genPrimes, hat_drei_Einsen n]  
->> [1117,1151,1171,1181,1511,1811,2111,...]
```

► Anwendung des G/F-Prinzips:

Alle Primzahlen mit Palindromdezimaldarstellung:

```
[ n | n <- genPrimes, ist_Palindrom n]  
->> [2,3,5,7,11,101,131,151,181,191,313,...]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# G/T-Prinzip am Bsp. des Primzahlstroms (1)

Ein Generator (G):

- ▶ `genPrimes :: [Integer]`  
`genPrimes = primes`

Viele Transformatoren (T):

- ▶ **Quadrieren** (für den Strom der Quadratprimzahlen):  
`square :: Integer -> Integer`  
`square n = ...`
- ▶ **Dekrementieren** (für den Strom der Primzahlvorgänger):  
`decrement :: Integer -> Integer`  
`decrement n = n-1`
- ▶ **Summieren** (für den Strom der partiellen Primzahlsummen (den Strom d. Summen d. Primzahlen von 2 bis  $n$ )):  
`sum :: [Integer] -> Integer`  
`sum ls = ...`
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# G/T-Prinzip am Bsp. des Primzahlstroms (2)

Zusammenfügen der G/T-Module zum Gesamtprogramm:

## ► Anwendung des G/T-Prinzips:

Der Strom der Quadratprimzahlen:

```
[ square n | n <- genPrimes ]  
->> [4,9,25,49,121,169,289,361,529,841,...]
```

## ► Anwendung des G/T-Prinzips:

Der Strom der Primzahlvorgänger:

```
[ decrement n | n <- genPrimes ]  
->> [1,2,4,6,10,12,16,18,22,28,...]
```

## ► Anwendung des G/T-Prinzips:

Der Strom der partiellen Primzahlsummen:

```
[ sum [2..n] | n <- genPrimes ]  
->> [2,5,14,27,65,90,152,189,275,434,...]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Terminierungsbemerkung

Auf **Terminierung** ist bei Anwendung der Prinzipien

► stets **besonders** zu achten.

So terminiert der Aufruf:

```
filter (<10) genPrimes ->> [2,3,5,7,
```

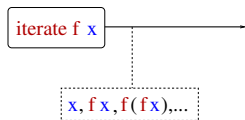
nicht; der Aufruf:

```
takeWhile (<10) genPrimes ->> [2,3,5,7]
```

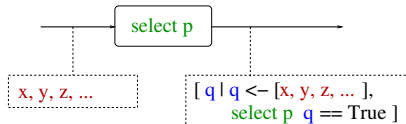
hingegen schon.

# Das G/S- und G/F-Prinzip auf einen Blick

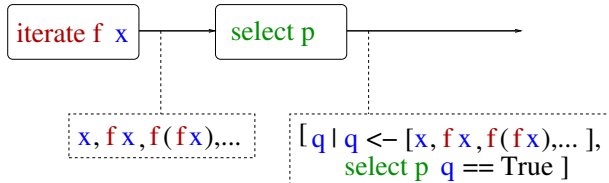
*Generator*



*Selektor/Filter*



*Verknüpfen von Generator und Selektor/Filter*



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

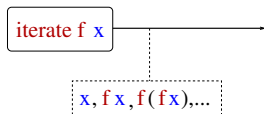
Teil V

Kap. 12

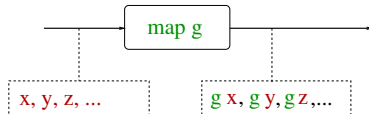
Kap. 13

# Das G/T-Prinzip auf einen Blick

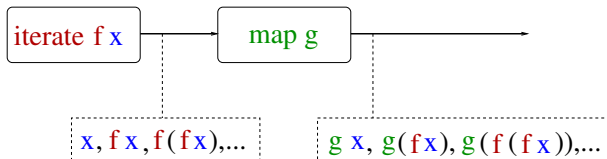
*Generator*



*Transformator*



*Verknüpfen von Generator und Transformator*



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Typische Anwendungen

...des G/S-, G/F- und G/T-Prinzips:

- ▶ Rucksackprobleme
- ▶ Pascalsches Dreieck
- ▶ Goldenes Verhältnis
- ▶ Fibonacci-Zahlen
- ▶ Potenzreihen
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Strom der Fibonacci-Zahlen mithilfe von map

...einfach, aber ineffizient (wg. der Ineffizienz von `fib`):

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

## Generator

```
fibs :: [Integer] -- Generator der Fibonacci-Zahlen
fibs = map fib [0..]
```

Strom von Argumenten

```
fibs ->> [0,1,1,2,3,5,8,13,21,34,55,...]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Fibonacci-Zahlen als Summe von Strömen (1)

Zwei **Generatoren** **G1** und **G2**:

**G1**, Strom der Fib.-Zahlen:      0 1 1 2 3 5 8 13 21...

**G2**, Rest d. Stroms d. Fib.-Z.: 1 1 2 3 5 8 13 21 34...

Summiere **G1** u. **G2**, '**G1+G2**':      + + + + + + + + ...

Rest des Restes des Stroms  
der Fibonacci-Zahlen      1 2 3 5 8 13 21 34 55...

Effiziente Berechnung der Fibonacci-Z. als Summe von **G1** u. **G2**:

**fibs** :: [Integer] -- Generator der Fibonacci-Zahlen

**fibs** = 0 : 1 : zipWith (+) **fibs** (tail **fibs**)

'Schopf'

'Sumpf' **G1**

**G2**

Rest d. Restes d. Stroms d. Fib.-Z.

Strom der Fibonacci-Zahlen

...sich wie Münchhausen 'am eigenen **Schopfe** aus dem **Sumpfe** ziehen'!

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Fibonacci-Zahlen als Summe von Strömen (2)

Aufruf von Generator `fibs`:

```
fibs ->> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...]
```

Aufrufe von Generator/Filter-, Selektorkombinationen mit `fibs`:

```
filter even fibs ->> [0,2,8,34,144, ...]
```

```
take 10 fibs ->> [0,1,1,2,3,5,8,13,21,34]
```

```
fibs!!5 ->> 3
```

Verwendete Hilfsfunktionen:

```
take :: Int -> [a] -> [a]
```

```
take 0 _ = []
```

```
take _ [] = []
```

```
take n (x:xs) | n>0 = x : take (n-1) xs
```

```
take _ _ = error "PreludeList.take: negative argument"
```

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

```
zipWith f _ _ = []
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Zusammenfassung

Späte Auswertung (engl. lazy evaluation) erlaubt es

- ▶ die Kontrolle der Auswertungsreihenfolge von Daten

zu trennen und ermöglicht dadurch die elegante Behandlung

- ▶ unendlicher Datenwerte (genauer: nicht a priori in der Größe beschränkter Datenwerte), insbesondere
  - ▶ unendlicher Listen, sog. Ströme (engl. streams, lazy lists)

Dies führt zu semantikbasierten, von der Programmlogik her begründeten neuen Modularisierungsprinzipien:

- ▶ Generator/Selektor-Prinzip
- ▶ Generator/Filter-Prinzip
- ▶ Generator/Transformator-Prinzip

sowie von Kombinationen dieser Prinzipien.

# Übungsaufgabe 18.2.1

Nicht nur **Generatoren** lassen sich wie in den Beispielen demonstrieren mit verschiedenen

- ▶ **Selektoren, Filtern, Transformatoren**

verknüpfen. Umgekehrt lassen sich auch **Selektoren, Filter** und **Transformatoren** mit verschiedenen

- ▶ **Generatoren**

verknüpfen.

Überlege und implementiere dazu einige Beispiele, die das demonstrieren.

# Kapitel 18.4

## Reflektives Programmieren

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

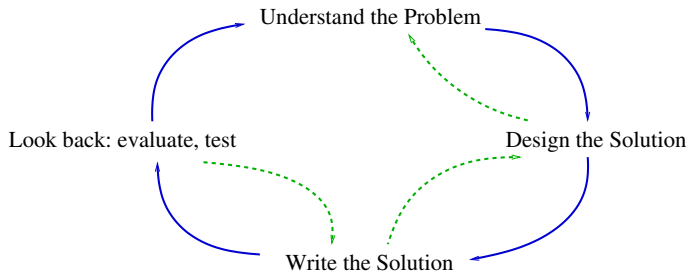
Teil V

Kap. 12

Kap. 13

# Reflektives Programmieren

...der Programm-Entwicklungszyklus nach Simon Thompson,  
Haskell: The Craft of Fuctional Programming, 2. Auflage,  
1999, Kap. 11 'Reflective Programming':



...in jeder der 4 Phasen ist es nützlich, (sich) Fragen zu stellen, zu beantworten und den Lösungsweg ggf. anzupassen.

Nosce te ipsum, nosce tuum opus.  
Erkenne dich selbst, erkenne dein Werk.

lat., sprichw., Apoll zugeschrieben, abgewandelt

# Phase 1: Typische Fragen

## Verstehen des Problems:

- ▶ Welches sind die Ein- und Ausgaben des Problems?
- ▶ Welche Randbedingungen sind einzuhalten?
- ▶ Ist das Problem über- oder underspezifiziert?
- ▶ Ist das Problem entscheidbar und damit grundsätzlich lösbar? In welche Komplexitätsklasse fällt es?
- ▶ Ist das Problem aufgrund seiner Struktur in Teilprobleme zerlegbar?
- ▶ ...



# Phase 2: Typische Fragen

## Entwerfen einer Lösung:

- ▶ Ist das Problem verwandt zu (mir) bekannten anderen, möglicherweise einfacheren Problemen?
- ▶ Wenn ja, lassen sich deren Lösungsideen anpassen und anwenden? Ebenso deren Implementierungen, vorhandene Bibliotheken?
- ▶ Lässt sich das Problem verallgemeinern und so möglicherweise sogar einfacher lösen?
- ▶ Ist das Problem mit den vorhandenen Ressourcen, einem gegebenen Budget lösbar?
- ▶ Ist die Lösung änderungs-, erweiterungs- und wiederbenutzungsfreundlich?
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Phase 3: Typische Fragen

## Ausformulieren und kodieren der Lösung:

- ▶ Gibt es passende Bibliotheken, speziell geeignete polymorphe Funktionen höherer Ordnung für die Lösung von Teilproblemen?
- ▶ Können vorhandene Bibliotheksfunktionen (zumindest) als Vorbild dienen, um entsprechende Funktionen für eigene Datentypen zu definieren?
- ▶ Kann funktionale Abstraktion (auch höherer Stufe) zur Verallgemeinerung der Lösung angewendet werden?
- ▶ Welche Hilfsfunktionen, Datenstrukturen könnten nützlich sein?
- ▶ Welche Möglichkeiten der Sprache können für die Codierung vorteilhaft ausgenutzt werden und wie?
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Phase 4: Typische Fragen

Blick zurück, evaluieren, testen:

- ▶ Lässt sich die Lösung testen, ihre Korrektheit beweisen, auch formal?
- ▶ Worin sind möglicherweise gefundene Fehler begründet? Flüchtigkeitsfehler, Programmierfehler, falsches oder unvollständiges Problemverständnis, falsches Semantikverständnis der verwendeten Programmiersprache? Andere Gründe?
- ▶ Sollte das Problem noch einmal gelöst werden müssen; sollte die Lösung und ihre Implementierung genauso gemacht werden? Was sollte beibehalten oder geändert werden und warum?
- ▶ Erfüllt das Programm auch nichtfunktionale Eigenschaften gut wie Performanz, Speicherverbrauch, Skalierbarkeit, Verständlichkeit, Änder- und Erweiterbarkeit?
- ▶ ...

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Am Ende gilt

Opus opificem probat.  
Das Werk empfiehlt den Meister.  
lat., sprichwörtl.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 18.5

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V




Kap. 12

Kap. 13





# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 18 (1)

-  Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2. Auflage, 1998. (Kapitel 9, Infinite Lists)
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 9, Infinite lists)
-  Richard Bird, Phil Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Kapitel 6.4, Divide and conquer; Kapitel 7, Infinite Lists)
-  Antonie J. T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 7.2, Infinite Objects; Kapitel 7.3, Streams)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 18 (2)

-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 2.2, Unendliche Datenstrukturen)
-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 14, Programming with Streams)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 15.6, Modular programming)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 18 (3)

-  Lambert Meertens. *Functional Pearl: Calculating the Sieve of Eratosthenes*. Journal of Functional Programming 14(6):759-763, 2004.
-  Matti Nykänen. *A Note on the Genuine Sieve of Eratosthenes*. Journal of Functional Programming 21(6):563-572, 2011.
-  Melissa E. O'Neill. *The Genuine Sieve of Eratosthenes*. Journal of Functional Programming 19(1):95-106, 2009.
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 20.2, Sortieren von Listen)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11


Teil V

Kap. 12

Kap. 13



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 18 (4)

-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 2, Faulheit währt unendlich)
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 8.1, Divide-and-conquer)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 12, Developing higher-order programs; Kapitel 17, Lazy programming)

# Teil VII

## Abschluss, Ausblick

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 19

## Abschließendes

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

Suavis est laborum praeteritorum memoria.  
Süss ist die Erinnerung an vergangene Mühen.

Cicero (106 - 43 v.Chr.)  
röm. Staatsmann und Schriftsteller

## Kapitel 19.1

### Rückblick

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Funktionale, imperative Programmierung (1)

Eigenschaften und Charakteristika im Vergleich.

## ► Funktional:

- Programm ist Ein-/Ausgaberation.
- Programme sind zustandsfrei und 'zeitlos'.
- Programmformulierung auf abstraktem, mathematisch geprägten Niveau, ohne eine Maschine im Blick.

## ► Imperativ:

- Programm ist Arbeitsanweisung für eine Maschine.
- Programme sind zustands- und 'zeitbehaftet'.
- Programmformulierung mit Blick auf eine Maschine, ein Maschinenmodell (von Neumann).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Funktionale, imperative Programmierung (2)

## ► Funktional:

- Die **Auswertungsreihenfolge** von Ausdrücken liegt (bis auf Datenabhängigkeiten) **nicht fest**.
- **Namen** werden durch **Wertvereinbarungen** **genau einmal** für immer an einen Wert **gebunden**.
- **Schachtelung (rekursiver) Funktionsaufrufe** erlaubt neue Werte mit neuen Namen zu verbinden.

## ► Imperativ:

- Die **Ausführungsreihenfolge** von Anweisungen liegt **fest**; Freiheiten bestehen bei der Auswertungsreihenfolge von Ausdrücken (wie funktional).
- **Namen** werden in der zeitlichen Abfolge durch **Zuweisungen temporär** mit Werten **belegt**.
- **Namen** können durch wiederholte Zuweisungen beliebig oft mit neuen Werten belegt werden (in **rekursiven Aufrufen**, **repetitiven Anweisungen** wie *while*, *repeat*, *for*).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Möglichkeiten fkt. Programmierung

*Die Fülle an Möglichkeiten  
[in funktionalen Programmiersprachen] erwächst  
aus einer kleinen Zahl von elementaren  
Konstruktionsprinzipien.*

Peter Pepper, *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-Verlag, 2. Auflage, 2003.

Im Fall von:

- ▶ **Funktionen**: (Fkt.-) Applikation, Fallunterscheidung, Rekursion.
- ▶ **Datenstrukturen**: Aufzählung, Produkt- und Summenbildung, Rekursion.

**Das Ganze ist mehr als die Summe seiner Teile.**

Aristoteles (384 - 322 v.Chr.)  
griech. Philosoph

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

1479/17

# Mächtigkeit fkt. Programmierung

...zusammen mit den durchgängigen Konzepten von

- ▶ Funktionen als **erstrangige Sprachelemente** (engl. first class citizens)
  - ▶ Funktionen höherer Ordnung
- ▶ **Polymorphie** auf
  - ▶ Funktionen
  - ▶ Datentypen

...führt dies zur Mächtigkeit und Eleganz **funktionaler Programmierung**, auf den Punkt gebracht im Slogan:

**Functional Programming is Fun!**



# Zur (rethorischen) Eingangsfrage

*Can programming be liberated  
from the von Neumann style?*

John W. Backus (1924-2007)  
*Turing Award* Preisträger 1977

Ja (im Detail kann diskutiert werden, siehe Ein-/Ausgabe).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Erfolgreiche Einsatzfelder fkt. Programmierung

- ▶ Theorembeweiser HOL und Isabelle in ML.
- ▶ Modellprüfer (z.B. Edinburgh Concurrency Workbench).
- ▶ Mobility Server von Ericson in Erlang.
- ▶ Konsistenzprüfung mit Pdiff (Lucent 5ESS) in ML.
- ▶ Übersetzer in übersetzter Sprache geschrieben.
- ▶ Datenbankabfragesprachen (z.B. CPL/Kleisli, in ML; Natural Expert, Haskell-ähnliche Abfragesprache).
- ▶ Protokollspezifikation (effiziente Fallstudien z.B. in ML).
- ▶ Expertensysteme (oft Lisp-basiert).
- ▶ ...
- ▶ <http://homepages.inf.ed.ac.uk/wadler/realworld>
- ▶ [www.haskell.org/haskellwiki/Haskell\\_in\\_industry](http://www.haskell.org/haskellwiki/Haskell_in_industry)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 12/17

# Rückblick auf die Vorbesprechung

...warum die nächste Sprache **funktional** sein sollte:

- ▶ Konrad Hinsen. **The Promises of Functional Programming**. Computing in Science and Engineering 11(4): 86-90, 2009.

...adopting a **functional programming** style **could make your programs more robust, more compact, and more easily parallelizable**.

- ▶ Konstantin Läufer, George K. Thiruvathukal. **The Promises of Typed, Pure, and Lazy Functional Programming: Part II**. Computing in Science and Engineering 11(5): 68-75, 2009.

...this second installment picks up where Konrad Hinsen's article "The Promises of Functional Programming" [...] left off, covering **static type inference** and **lazy evaluation** in **functional programming languages**.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Haben Sie eine Antwort für sich gefunden?

...ich bin (als Programmierer) ein Typ

- ▶ klarer Ansagen, denke imperativ in Instruktionen und ihren Effekten und handle als Instrukteur.



- ▶ harmonischen, konsensualen Ausgleichs, denke funktional in Gleichungssystemen und Eigenschaften ihrer Lösungen und handle als Delegateur.



- ▶ faszinierender Empathie- und Herzlosigkeit, denke logisch in Formeln und ihren logischen Konsequenzen und handle als Spockteur.



Ich denke nie an die Zukunft.  
Sie kommt früh genug.

Albert Einstein (1879-1955)  
dt.-schweiz.-amerik. Physiker

Wer nicht an die Zukunft denkt,  
wird bald Sorgen haben.

Konfuzius (551 - 479 v.Chr.)  
chin. Ethiker und Staatslehrer

## Kapitel 19.2

### Ausblick

Die Zukunft hat schon begonnen.

Robert Jungk (1913-1994)  
österr. Wissenschaftspublizist und Zukunftsforscher

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17

Alles, was man wissen muss,  
um selber weiter zu lernen,  
...ist jetzt gelernt.

Dietrich Schwanitz (1940-2004)  
dt. Anglistikprof. und Schriftsteller  
(verkürzt und ergänzt in seinem Sinn)

Fort- und weiterführendes zu funktionaler Programmierung in  
TUW-Lehrveranstaltungen, insbesondere:

- ▶ LVA 185.A05 Fortgeschrittene funktionale Programmierung, VU 2.0, ECTS 3.0.
- ▶ LVA 183.653 Methodisches, industrielles Software-Engineering mit funktionalen Sprachen am Fallbeispiel von Haskell, VU 2.0, ECTS 3.0, ao.Prof. Thomas Grechenig.
- ▶ LVA 127.008 Haskell-Praxis: Programmieren mit der funktionalen Programmiersprache Haskell, VU 2.0, ECTS 3.0, Prof. em. Andreas Frank, Institut für Geoinformation und Kartographie.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

Man muss so lange lernen,  
wie man etwas nicht weiß;  
und wenn wir dem Sprichwort glauben,  
ein Leben lang.

Seneca der Jüngere (um 4 v.Chr. - 65 n.Chr.)  
röm. Politiker, Philosoph und Schriftsteller  
Epistulae ad Lucilium 76,1-4

## Lebenslanges Lernen?

Nihil novi sub sole.  
Es gibt nicht Neues unter der Sonne.

Vulgata, lat. Neuübersetzung der Bibel aus dem 4. Jhdt. von  
Hieronymus (um 347 - 419 oder 420)  
Liber Ecclesiastes 1,10

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## Vorlesungsinhalte:

- ▶ **Programmieren** mit
  - ▶ Strömen, Funktoren, Monaden, Kombinatorbibliotheken.
  - ▶ Funktionalen Feldern, abstrakten Datentypen.
- ▶ **Anwendungen**
  - ▶ Funktionale Perlen, Algorithmenmuster, funktionale reaktive Programmierung, logische Programmierung funktional, Strukturanalyse (engl. Parsing).
- ▶ **Qualitätssicherung**
  - ▶ Programmverifikation, Programmvalidation, gleichungsbasiertes Schließen und Beweisen, automatisches Testen.
- ▶ ...



## Vorlesungsinhalte:

- ▶ **Analyse** und **Verbesserung** von gegebenem Code.
- ▶ **Weiterentwicklung** der Open-Source-Entwicklungsumgebung **LEKSAH** für Haskell, insbesondere der graphischen Benutzerschnittstelle (GUI).
- ▶ **Gestaltung** graphischer Benutzerschnittstellen (GUIs) mit **Glade** und **Gtk+**.
- ▶ ...

# Always look on the bright side of life

*The clarity and economy of expression that the language of functional programming permits is often very impressive, and, but for human inertia, functional programming can be expected to have a brilliant future.*<sup>(\*)</sup>

Edsger W. Dijkstra (1930-2002)  
Turing Award Preisträger 1972

<sup>(\*)</sup> Zitat aus: Introducing a course on calculi. Ankündigung einer Lehrveranstaltung an der University of Texas at Austin, 1995.

Der größte Feind des Fortschritts  
ist nicht der Irrtum, sondern die Trägheit.

Henry Thomas Buckle (1821-1862)  
engl. Historiker, aus "Geschichte der Zivilisation" (unvollendet)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe 19.2.1 – Eiskunstlauf (1)

Eiskunstlauf heißt vor allem Disziplin,  
Willensstärke und Ausdauer.  
Einen neuen Sprung richtig zu beherrschen,  
kann schon mal ein Jahr dauern.  
Oder auch etwas länger.

Sophia Schaller (\* 2000)  
österr. Eiskunstläuferin

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe 19.2.1 – Philosophie (2)

Die Kenntnis der Philosophie haben  
die Götter niemandem mitgegeben,  
doch die Fähigkeit allen.

Hätten die Götter auch die Philosophie  
zu einem Allgemeingut gemacht und würden wir  
bereits einsichtig geboren, hätte die Weisheit  
ihren größten Vorzug verloren, nämlich  
nicht zu den Zufallsgaben zu gehören.

Nun freilich ist gerade dies an ihr so kostbar und  
herrlich, dass sie einem nicht einfach zufällt, sondern  
dass jeder sie nur sich selbst verdankt, dass sie nicht  
von einem anderen erbeten werden kann.

Warum solltest du zur Philosophie aufblicken, wenn  
sie eine Sache der Gefälligkeit wäre?

Seneca der Jüngere (um 4 v.Chr. – 65 n.Chr.)  
röm. Politiker, Philosoph und Schriftsteller  
Epistulae ad Lucilium 90,1-3

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe 19.2.1 – Bergsteigen (3)

Man kann sich keinen Gipfel erkaufen.

Peter Habeler (\* 1942)  
österr. Extrembergsteiger

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe 19.2.1 – Informatik (4)

Ersetze in den Zitaten von [Sophia Schaller](#), [Seneca](#) und [Peter Habeler](#)

- ▶ [Eiskunstlauf](#), [Philosophie/Weisheit](#) und [Gipfel](#) durch [Informatik](#)

und

- ▶ [neuen Sprung](#) durch [neues Paradigma](#) und [funktionale Programmierung](#).

Ergeben sich erneut sinnvolle, bedenkenswerte Aussagen? Sind sie nachvollziehbar?

# Übungsaufgabe 19.2.2 – Schätzaufgabe (1)

Ein **ECTS**-Punkt entspricht einem erwarteten ernsthaften (ungleich gefühltem) Arbeitsaufwand im Ausmaß von

- ▶ 25 Stunden in Italien, Österreich, Spanien,...
- ▶ 26 Stunden in Estland,...
- ▶ 27 Stunden in Finnland,...
- ▶ 28 Stunden in Dänemark, Portugal,...
- ▶ 30 Stunden in Deutschland, Rumänien, Schweiz,...
- ▶ ...

# Übungsaufgabe 19.2.2 – Schätzaufgabe (2)

1. Schätze, mit wie vielen ECTS-Punkten der Aufwand des
  - 1.1 'zu richtiger Beherrschung' führenden Lernens und Einübens eines neuen Sprungs im Eiskunstlauf
  - 1.2 'zu richtiger Beherrschung' führenden Lernens und Einübens des funktionalen Programmierstilsin Österreich, der Schweiz angemessen bewertet wäre?
2. Wie liegen diese Schätzwerte im Vergleich zum Referenzwert von 3 ECTS-Punkten für diese Lehrveranstaltung entsprechend 75 Aufwandsstunden für in Österreich Studierende bzw. 90 Aufwandsstunden für in der Schweiz Studierende? Höher? Niedriger?
3. Welche Schlussfolgerungen können aus diesem Vergleich gezogen, welche Maßnahmen ggf. abgeleitet werden?



# Übungsaufgabe 19.2.3 – Feldstudie

Beobachten Sie Ihre Umgebung. Gibt es Parallelen zum Studium?

Um erfolgreich zu sein,  
brauchst du das ganze Paket:  
Talent, mehr Fleiß als die anderen,  
Risikobereitschaft, keine Angst  
vor der Niederlage.

Franz Klammer (\* 1953)  
österr. Skirennläufer, Olympiasieger 1976

Italien ist wie ein Athlet mit  
Potenzial, der keine Veranlassung sieht,  
an sich zu arbeiten. Ich hasse die  
italienische Mentalität. Wir wollen  
mit dem geringstmöglichen Aufwand  
das Maximum erreichen.

Sofia Goggia (\* 1992)  
ital. Skirennläuferin, Olympiasiegerin 2018

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

In manchen Phasen muss man die  
Zähne zusammenbeißen: Stürze  
tun weh [ergänzt: Niederlagen auch].

Sophia Schaller (\* 2000)  
österreich. Eiskunstläuferin

...gleich ob Eiskunstlauf, Philosophie, Informatik,...

# Ewige Wahrheiten

Scientia prodest.  
Wissen nützt.

Cicero (106 - 43 v.Chr.)  
röm. Staatsmann und Schriftsteller

Alle Menschen streben  
von Natur aus nach Wissen.

Aristoteles (384 - 322 v.Chr.)  
griech. Philosoph

Die Summe unserer Erkenntnis  
besteht aus dem, was wir gelernt,  
und aus dem, was wir vergessen haben.

Marie von Ebner-Eschenbach (1830-1916)  
österreich. Schriftstellerin

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Glückes Geschick

Semper aliquid haeret.  
Etwas bleibt immer hängen.

Plutarch (um 46 - nach 119 n.Chr.)  
griech. Schriftsteller

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Kapitel 19.3

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 19 (1)



Simon Peyton Jones. *16 Years of Haskell: A Retrospective on the occasion of its 15th Anniversary – Wearing the Hair Shirt: A Retrospective on Haskell*. Invited Keynote Presentation at the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03), 2003.

[research.microsoft.com/users/simonpj/papers/haskell-retrospective/](http://research.microsoft.com/users/simonpj/papers/haskell-retrospective/)



Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. *A History of Haskell: Being Lazy with Class*. In Proceedings of the 3rd ACM SIGPLAN 2007 Conference on History of Programming Languages (HOPL III), 12-1 - 12-55, 2007. (ACM Digital Library [www.acm.org/dl](http://www.acm.org/dl))

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V



Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 19 (2)





-  Andrew Appel. *A Critique of Standard ML*. Journal of Functional Programming 3(4):391-430, 1993.
-  Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988. (Kapitel 5, Alternative functional styles)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 1.3, Features of Haskell; Kapitel 1.4, Historical background)
-  Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. eXamen.press, 2009. (Kapitel 3, Programmiersprachen)

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 19 (3)

-  Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011. (Kapitel 1, Introduction; Kapitel 9, Functional programming in Standard ML; Kapitel 10, Functional programming and LISP)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 23, Compiler and Interpreter für Opal, ML, Haskell, Gofer)
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 1.2, Functional Languages)
-  Colin Runciman, David Wakeling. *Applications of Functional Programming*. UCL Press, 1995.



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Kapitel 19 (4)

-  Dietrich Schwanitz. *Bildung: Alles, was man wissen muss*. Eichborn Verlag, 1999.
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999.  
(Anhang A, Functional, imperative and OO programming)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.  
(Anhang A, Functional, imperative and OO programming)
-  Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.

# Literaturverzeichnis

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

I cannot live without books.

Thomas Jefferson (1743-1826)

amerik. Staatsmann und Philosoph

3. Präsident der USA

Hauptverfasser der amerik. Unabhängigkeitserklärung

## Literaturhinweise und Leseempfehlungen

...zum vertiefenden und weiterführenden Selbststudium.

- ▶ I Lehrbücher
- ▶ II Tutorien, Manuale
- ▶ III Grundlegende, wegweisende Artikel
- ▶ IV Weitere Arbeiten
- ▶ V Zum Haskell-Sprachstandard
- ▶ VI Die Haskell-Geschichte

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10







Kap. 11

Teil V

Kap. 12

Kap. 13

# I Lehrbücher (1)

-  Christopher Allen, Julie Moronuki. *Haskell Programming from First Principles*. ebook. <http://haskellbook.com>.
-  Henri E. Baal, Dick Grune. *Programming Language Essentials*. Addison-Wesley, 1994.
-  Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised Edn., North-Holland, 1984.
-  Henrik P. Barendregt, Wil Dekkers, Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2012.
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015.
-  Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2. Auflage, 1998.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10






Kap. 11

Teil V

Kap. 12

Kap. 13

# I Lehrbücher (2)

-  Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.
-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011.
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004.
-  Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein. *Algorithmen – Eine Einführung*. Oldenbourg Verlag, 2004.
-  Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# I Lehrbücher (3)

-  Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012.
-  Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004.
-  Gilles Dowek, Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Springer-V., 2011.
-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999.
-  Anthony J. Field, Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10






Kap. 11

Teil V

Kap. 12

Kap. 13

# I Lehrbücher (4)

-  Matthias Felleisen, Rober B. Findler, Matthew Flatt, Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.
-  Hugh Glaser, Chris Hankin, David Till. *Principles of Functional Programming*. Prentice Hall, 1984.
-  Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004.
-  Peter Henderson. *Functional Programming: Application and Implementation*. Prentice Hall, 1980.
-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# I Lehrbücher (5)

-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016.
-  Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. eXamen.press, 2009.
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011.  
[learnyouahaskell.com](http://learnyouahaskell.com)
-  Bruce J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley, 1990.
-  Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11







Teil V

Kap. 12

Kap. 13



# I Lehrbücher (6)

-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. [book.realworldhaskell.org](http://book.realworldhaskell.org)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003.
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006.
-  Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009.
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999.
-  Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# I Lehrbücher (7)

-  Peter Rechenberg, Gustav Pomberger (Hrsg.). *Informatik-Handbuch*. Carl Hanser Verlag, 4. Auflage, 2006.
-  Colin Runciman, David Wakeling. *Applications of Functional Programming*. UCL Press, 1995.
-  Steven S. Skiena. *The Algorithm Design Manual*. Springer-V., 1998.
-  Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik. Induktives Vorgehen*. Springer-V., 2014.
-  Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# I Lehrbücher (8)

-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999.
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.
-  Allen B. Tucker (Editor-in-Chief). *Computer Science Handbook*. Chapman & Hall/CRC, 2004.
-  Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008.
-  Reinhard Wilhelm, Helmut Seidl. *Compiler Design – Virtual Machines*. Springer-V., 2010.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V

Kap. 12

Kap. 13

## II Tutorien, Manuale (1)

-  H. Conrad Cunningham. *Notes on Functional Programming with Haskell*. Course Notes, University of Mississippi, 2007. [citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf)
-  Hal Daumé III. *Yet Another Haskell Tutorial*. wikibooks.org-Ausgabe, 2007. [https://en.wikibooks.org/wiki/Yet\\_Another\\_Haskell\\_Tutorial](https://en.wikibooks.org/wiki/Yet_Another_Haskell_Tutorial)
-  Chris Done. *Try Haskell*. Online Hands-on Haskell Tutorial. [tryhaskell.org](http://tryhaskell.org).
-  Paul Hudak, Joseph Fasel, John Peterson. *A Gentle Introduction to Haskell*. Technischer Bericht, Yale University, 1996. <https://www.haskell.org/tutorial>

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17

## II Tutorien, Manuale (2)



Hugs-Benutzerhandbuch. *The Hugs98 User Manual*.

<https://www.haskell.org/hugs/pages/hugsman/index.html>



GHCI-Benutzerhandbuch. *Glasgow Haskell Compiler User's Guide*. [http://www.haskell.org/ghc/docs/latest/html/users\\_guide/ghci.html](http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci.html)



Haskells Standard-Präludium.

<https://www.haskell.org/onlinereport/standard-prelude.html>

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V

Kap. 12

Kap. 13

# III Grundlegende, wegweisende Artikel (1)

-  John W. Backus. *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*. Communications of the ACM 21(8):613-641, 1978.
-  Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941.
-  Robert W. Floyd. *The Paradigms of Programming*. Turing Award Lecture. Communications of the ACM 22(8):455-460, 1979.
-  John Hughes. *Why Functional Programming Matters*. The Computer Journal 32(2):98-107, 1989.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# III Grundlegende, wegweisende Artikel (2)

-  Paul Hudak. *Conception, Evolution and Applications of Functional Programming Languages*. Communications of the ACM 21(3):359-411, 1989.
-  Christopher Strachey. *Fundamental Concepts in Programming Languages*. Higher-Order and Symbolic Computation 13:11-49, 2000, Kluwer Academic Publishers (revised version of a report of the NATO Summer School in Programming, Copenhagen, Denmark, 1967.)
-  Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# IV Weitere Arbeiten (1)

-  Wilhelm Ackermann. *Zum Hilbertschen Aufbau der reellen Zahlen*. Mathematische Annalen 99:118-133, 1928.
-  Andrew Appel. *A Critique of Standard ML*. Journal of Functional Programming 3(4):391-430, 1993.
-  Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, Philip Wadler. *The Call-by-Need Lambda Calculus*. In Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), 233-246, 1995.
-  Hendrik Pieter Barendregt, Erik Barendsen. *Introduction to the Lambda Calculus*. Revised Edn., Technical Report, University of Nijmegen, March 2000.  
<ftp://ftp.cs.kun.nl/pub/CompMath.Found/lambda.pdf>

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11




Teil V

Kap. 12

Kap. 13



## IV Weitere Arbeiten (2)

-  Luca Cardelli. *Basic Polymorphic Type Checking*. Science of Computer Programming 8:147-172, 1987.
-  B. Jack Copeland, Oron Shagrir. *The Church-Turing Thesis: Logical Limit or Breachable Barrier?* Communications of the ACM 62(1):66-74, 2019.
-  Iavor S. Dachki, Thomas Hallgren, Mark P. Jones, Rebekah Leslie, Andrew Tolmach. *Writing System Software in a Functional Language: An Experience Report*. In Proceedings of the 4th International Workshop on Programming Languages and Operating Systems (PLOS 2007), Article No. 1, 5 pages, 2007.

## IV Weitere Arbeiten (3)



Luís Damas, Robin Milner. *Principal Type Schemes for Functional Programming Languages*. In Conference Record of the 9th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82), 207-218, 1982.



Noah M. Daniels, Andrew Gallant, Norman Ramsey. *Experience Report: Haskell in Computational Biology*. In Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012), 227-234, 2012.



Frank DeRemer, Hans H. Kron. *Programming-in-the-Large vs. Programming-in-the-Small*. IEEE Transactions on Software Engineering 2(2):80-86, 1976.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## IV Weitere Arbeiten (4)

-  Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *The Architecture of the Utrecht Haskell Compiler*. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009)*, 93-104, 2009.
-  Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *UHC Utrecht Haskell Compiler*, 2009. [www.cs.uu.nl/wiki/UHC](http://www.cs.uu.nl/wiki/UHC)
-  Neal Ford. *Functional Thinking: Why Functional Programming is on the Rise*. IBM developerWorks, 11 pages, 2013. [www.ibm.com/developerworks/java/library/j-ft20/j-ft20-pdf.pdf](http://www.ibm.com/developerworks/java/library/j-ft20/j-ft20-pdf.pdf)
-  Robert M. French. *Moving Beyond the Turing Test*. *Communications of the ACM* 55(12):74-77, 2012.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V

Kap. 12

Kap. 13

## IV Weitere Arbeiten (5)

-  Robin Gandy. *The Confluence of Ideas in 1936*. In Rolf Herken (Hrsg.), *The Universal Turing Machine: A Half-Century Survey*. Springer-V., 2. Auflage, 51-102, 1995.
-  Hugh Glaser, Pieter H. Hartel, Paul W. Garrat. *Programming by Numbers: A Programming Method for Novices*. *The Computer Journal* 43(4):252-265, 2000.
-  Benjamin Goldberg. *Functional Programming Languages*. *ACM Computing Surveys* 28(1):249-251, 1996.
-  Andrew J. Gordon. *Functional Programming and Input/Output*. *British Computer Society Distinguished Dissertations in Computer Science*. Cambridge University Press, 1994.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10






Kap. 11

Teil V

Kap. 12

Kap. 13

## IV Weitere Arbeiten (6)

-  John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM 20(6):396-404, 1977.
-  John V. Guttag, James J. Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica 10(1):27-52, 1978.
-  John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.
-  Bastiaan Heeren, Daan Leijen, Arjan van IJzendoorn. *Helium, for Learning Haskell*. In Proceedings of the ACM SIGPLAN 2003 Haskell Workshop (Haskell 2003), 62-71, 2003.
-  Konrad Hinsén. *The Promises of Functional Programming*. Computing in Science and Engineering 11(4):86-90, 2009.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10







Kap. 11

Teil V

Kap. 12

Kap. 13

## IV Weitere Arbeiten (7)

-  C.A.R. Hoare. *Algorithm 64: Quicksort*. Communications of the ACM 4(7):321, 1961.
-  C.A.R. Hoare. *Quicksort*. The Computer Journal 5(1):10-15, 1962.
-  Ian Horswill. *What is Computation?* Crossroads, the ACM Magazine for Students 18(3):8-14, 2012.
-  Paul Hudak, Joseph H. Fasel. *A Gentle Introduction to Haskell*. ACM SIGPLAN Notices 27(5):1-52, 1992.
-  Arjan van IJzendoorn, Daan Leijen, Bastiaan Heeren. *The Helium Compiler*. [www.cs.uu.nl/helium](http://www.cs.uu.nl/helium).
-  Neil D. Jones. *Constant Time Factors do Matter*. In Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC'93), 602-611, 1993.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10






Kap. 11

Teil V

Kap. 12

Kap. 13

## IV Weitere Arbeiten (8)

-  Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. Computing in Science and Engineering 1(3):64-72, 1999.
-  Stephen C. Kleene. *General Recursive Functions of Natural Numbers*. Mathematische Annalen 112:727-742, 1936.
-  Stephen C. Kleene.  $\lambda$ -Definability and Recursiveness. Duke Mathematical Journal 2:340-352, 1936.
-  Stephen C. Kleene. *Origins of Recursive Function Theory*. Annals of the History of Computing 3:52-67, 1981.
-  Donald E. Knuth. *Big Omicron and Big Omega and Big Theta*. ACM SIGACT News 8(2):18-24, 1976.  
(s.a. Nachdruck unter gleichem Titel in: Donald E. Knuth. *Selected Papers on Analysis of Algorithms*. CSLI Lecture Notes Number 102, CSLI Publications, 35-41, 2012.)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V

Kap. 12

Kap. 13

## IV Weitere Arbeiten (9)

-  Donald E. Knuth. *Literate Programming*. The Computer Journal 27(2):97-111, 1984.
-  Konstantin Läufer, George K. Thiruvathukal. *The Promises of Typed, Pure, and Lazy Functional Programming: Part II. Computing in Science and Engineering* 11(5):68-75, 2009.
-  John Maraist, Martin Odersky, David N. Turner, Philip Wadler. *Call-by-name, Call-by-value, call-by-need, and the Linear Lambda Calculus*. Electronic Notes in Theoretical Computer Science 1:370-392, 1995.
-  John Maraist, Martin Odersky, Philip Wadler. *The Call-by-Need Lambda Calculus*. Journal of Functional Programming 8(3):275-317, 1998.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11





Teil V

Kap. 12

Kap. 13



## IV Weitere Arbeiten (10)

-  John Maraist, Martin Odersky, David N. Turner, Philip Wadler. *Call-by-name, Call-by-value, call-by-need, and the Linear Lambda Calculus*. *Theoretical Computer Science* 228(1-2):175-210, 1999.
-  Mihai Maruseac. *Haskell: A Language for Modern Times*. *Crossroads, the ACM Magazine for Students* 24(1):64-66, 2017.
-  John McCarthy. *A Basis for a Mathematical Theory of Computation*. In *Computer Programming and Formal Systems*, Paul Braffort, David Hirschberg (Hrsg.), North-Holland, 33-70, 1963.
-  Lambert Meertens. *Functional Pearl: Calculating the Sieve of Eratosthenes*. *Journal of Functional Programming* 14(6):759-763, 2004.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10






Kap. 11

Teil V

Kap. 12

Kap. 13

## IV Weitere Arbeiten (11)

-  Donald Michie. *'Memo' Functions and Machine Learning*. Nature 218:19-22, 1968.
-  Robin Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences 17:248-375, 1978.
-  Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.
-  John C. Mitchell. *Type Systems for Programming Languages*. In *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*, Jan van Leeuwen (Hrsg.). Elsevier Science Publishers, 367-458, 1990.
-  William Newman. *Alan Turing Remembered – A Unique Firsthand Account of Formative Experiences with Alan Turing*. Communications of the ACM 55(12):39-41, 2012.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V






Kap. 12

Kap. 13

## IV Weitere Arbeiten (12)

-  **Matti Nykänen.** *A Note on the Genuine Sieve of Eratosthenes.* *Journal of Functional Programming* 21(6):563-572, 2011.
-  **Melissa E. O'Neill.** *The Genuine Sieve of Eratosthenes.* *Journal of Functional Programming* 19(1):95-106, 2009.
-  **David L. Parnas.** *On the Criteria to be used on Decomposing Systems into Modules.* *Communications of the ACM* 15(12):1053-1058, 1972.
-  **David L. Parnas, Paul C. Clements, David M. Weiss.** *The Modular Structure of Complex Systems.* *IEEE Transactions on Software Engineering* 11(3):259-266, 1985.

## IV Weitere Arbeiten (13)

-  Rózsa Péter. *Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktionen.* *Mathematische Annalen* 110:612-632, 1934.
-  Rózsa Péter. *Konstruktion nichtrekursiver Funktionen.* *Mathematische Annalen* 111:42-60, 1935.
-  Gordon Plotkin. *Call-by-name, Call-by-value, and the  $\lambda$ -Calculus.* *Theoretical Computer Science* 1:125-159, 1975.
-  Norman Ramsey. *On Teaching How to Design Programs.* In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP 2014)*, 153-166, 2014.
-  J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle.* *Journal of the ACM* 12(1):23-42, 1965.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10


Kap. 11

Teil V

Kap. 12

Kap. 13

## IV Weitere Arbeiten (14)

-  Chris Sadler, Susan Eisenbach. *Why Functional Programming?* In *Functional Programming: Languages, Tools and Architectures*, Susan Eisenbach (Hrsg.), Ellis Horwood, 9-20, 1987.
-  Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.
-  Uwe Schöning, Wolfgang Thomas. *Turings Arbeiten über Berechenbarkeit – eine Einführung und Lesehilfe*. Informatik Spektrum 35(4):253-260, 2012.
-  Curt J. Simpson. *Experience Report: Haskell in the “Real World”: Writing a Commercial Application in a Lazy Functional Language*. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13/17

# Weitere Arbeiten (15)

-  Michael Snoyman. *Developing Web Applications with Haskell and Yesod*. O'Reilly, 2012.
-  Simon Thompson. *Where Do I Begin? A Problem Solving Approach in Teaching Functional Programming*. In *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97)*, Springer-Verlag, LNCS 1292, 323-334, 1997.
-  Boris A. Trakhtenbrot *Comparing the Church and Turing Approaches: Two Prophetic Messages*. In Rolf Herken (Hrsg.), *The Universal Turing Machine: A Half-Century Survey*. Springer-V., 2. Auflage, 557-582, 1995.
-  Moshe Vardi. *Self-Reference and Section 230*. *Communications of the ACM* 61(11):7, 2018.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10






Kap. 11

Teil V

Kap. 12





Kap. 13

# Weitere Arbeiten (16)

-  Alvaro Videla. *Metaphors We Compute By*. Communications of the ACM 60(10):42-45, 2017.
-  Philip Wadler. *An angry half-dozen*. ACM SIGPLAN Notices 33(2):25-30, 1998.
-  Philip Wadler. *Why no one uses Functional Languages*. ACM SIGPLAN Notices 33(8):23-27, 1998.
-  Philip Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science 2:461-493, 1992.
-  Interview mit John Hughes über 'Funktionale Programmierung und Haskell'.  
<https://www.youtube.com/watch?v=LnX3B9oaKzw>

# Weitere Arbeiten (17)

Welches Paradigma, welche Sprache sollte ich nutzen?

-  Peter J. Landin. *The next 700 Programming Languages*. Communications of the ACM 9(3):157-166, 1966.
-  Jeffrey S. Foster. *Shedding New Light on an Old Language Debate*. Communications of the ACM 60(10):90, 2017.
-  Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, Vladimir Filkov. *A Large-Scale Study of Programming Languages and Code Quality in GitHub*. Communications of the ACM 60(10):91-100, 2017.
-  Rachel Harrison, L. G. Smaraweera, Mark R. Dobie, Paul H. Lewis. *Comparing Programming Paradigms: An Evaluation of Functional and Object-Oriented Programs*. Software Engineering Journal 11(4):247-254, 1996.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# V Zum Haskell-Sprachstandard



Paul Hudak, Simon Peyton Jones, Philip Wadler (Hrsg.). *Report on the Programming Language Haskell: Version 1.1*. Technical Report, Yale University and Glasgow University, August 1991.



Paul Hudak, Simon Peyton Jones, Philip Wadler (Hrsg.). *Report on the Programming Language Haskell: A Non-strict Purely Functional Language (Version 1.2)*. ACM SIGPLAN Notices, 27(5):1-164, 1992.



Simon Marlow (Hrsg.). *Haskell 2010 Language Report*, 2010.  
[www.haskell.org/definition/haskell2010.pdf](http://www.haskell.org/definition/haskell2010.pdf)



Simon Peyton Jones (Hrsg.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. [www.haskell.org/definitions](http://www.haskell.org/definitions).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## VI Die Haskell-Geschichte



Simon Peyton Jones. *16 Years of Haskell: A Retrospective on the occasion of its 15th Anniversary – Wearing the Hair Shirt: A Retrospective on Haskell*. Invited Keynote Presentation at the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003), 2003.

[research.microsoft.com/users/simonpj/papers/haskell-retrospective/](http://research.microsoft.com/users/simonpj/papers/haskell-retrospective/)



Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. *A History of Haskell: Being Lazy with Class*. In Proceedings of the 3rd ACM SIGPLAN 2007 Conference on History of Programming Languages (HOPL III), 12-1 - 12-55, 2007. (ACM Digital Library [www.acm.org/dl](http://www.acm.org/dl))

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

Daher verehere ich die Erkenntnisse der  
Weisheit und ihre Entdecker; mit Freude nähere  
ich mich gleichsam dem Vermächtnis vieler.  
Für mich ist dies alles erworben, für mich erarbeitet.  
Doch wir wollen uns als guter Familienvater erweisen  
und mehr hinterlassen, als wir bekommen haben;  
dies Erbe soll vergrößert von mir auf die Nachwelt  
übergehen. Viel bleibt noch zu tun, und viel  
wird bleiben, und keinem, der nach tausend  
Menschenaltern auf die Welt kommt, ist die  
Möglichkeit versperrt, noch etwas hinzuzufügen.

Aber selbst wenn schon alles von den alten Denkern herausgefunden  
wurde, wird dies immer neu sein: die Anwendung, das Verstehen  
und die Neuordnung der von anderen stammenden Erkenntnisse.

Seneca der Jüngere (um 4 v.Chr. - 65 n.Chr.)  
röm. Politiker, Philosoph und Schriftsteller  
Epistulae ad Lucilium 64, 7f.

Seneca. Der Weise ist sich selbst genug. Gedanken für alle Lebenslagen.  
Übersetzt u. herausgegeben von Ursula Blank-Sangmeister, Reclam, 2014.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

Wenn ich weiter gesehen habe als andere,  
so deshalb, weil ich auf den Schultern von Riesen stehe.

gewöhnlich Isaac Newton zugeschrieben,  
jedoch wesentlich älteren Ursprungs  
Sir Isaac Newton (1643-1727)  
engl. Naturforscher, Physiker und Philosoph

# Anhänge

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# A

## Imperative vs. funktionale Programmierung: Schlaglichter

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# A.1

## Programmatischer Kern

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Imperativ vs. fkt.: Programmatischer Kern

In **imperativen** (prozeduralen, objektorientierten) **Programmen** geht es um die **Ausführung** von **Anweisungen** (dafür müssen auch **Ausdrücke** ausgewertet werden):

```
a := y + z;  
b := x + y;  
if a>0 then c := a*b else c := a/2;
```

In **funktionalen Programmen** geht es ausschließlich um die **Auswertung** von **Ausdrücken**. Anweisungen gibt es nicht!

```
y + z  
x + y  
if (y+z)>0 then (y+z)*(x+y) else (y+z)/2
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Imperativ: Anweisungen und Ausdrücke

...im Detail:

a := y + z;  
*Ausdruck*  
*Anweisung*

b := x + y;  
*Ausdruck*  
*Anweisung*

if a > 0 then c := a \* b else c := a / 2;  
*Ausdruck*      *Ausdruck*      *Ausdruck*  
                  *Anweisung*      *Anweisung*  
*Anweisung*

Genauer sind auch 0 und 2 Ausdrücke; y, z, x, a, b und c hingegen Programmvariable, deren Wert abseits von linksseitigen Vorkommen als Ausdruck verwendet wird.

# Funktional: Ausschließlich Ausdrücke

...im Detail:

$y + z$   
Ausdruck

$x + y$   
Ausdruck

if  $(y+z) > 0$  then  $(y+z) * (x+y)$  else  $(y+z) / 2$   
Ausdruck      Ausdruck      Ausdruck      Ausdruck  
Ausdruck      Ausdruck

Genauer sind auch  $y$ ,  $z$ ,  $x$ ,  $0$  und  $2$  Ausdrücke.

# A.2

## Namensvereinbarungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## A.2.1

# Namensvereinbarungen in imperativen Programmen

# Namensvereinbarungen

...werden getroffen u.a. für:

- ▶ Variablen: Variablennamen
- ▶ Prozeduren/Methoden: Prozedur-/Methodennamen
- ▶ Funktionen: Funktionsnamen
- ▶ Typen: Typnamen
- ▶ ...

Dabei gilt: Die Einführung benannter

- ▶ Variablen, Prozeduren/Methoden ist von überragender Bedeutung.
- ▶ Funktionen ist oft nur eingeschränkt möglich.
- ▶ Typen, etc.: Für diesen Abschnitt nicht relevant und daher unbetrachtet bleibend.

## A.2.2

# Namensvereinbarungen in funktionalen Programmen

# Namensvereinbarungen

Für Ausdrücke können **Namen** vergeben, **vereinbart** werden:

$$a = y + z$$

Vereinb. von Name  $a$  für Ausdruck  $y + z$

$$b = x + y$$

Vereinb. von Name  $b$  für Ausdruck  $x + y$

$$c = \text{if } (y+z)>0 \text{ then } (y+z)*(x+y) \text{ else } (y+z)/2$$

Vereinb. von Name  $c$  für Ausdr.  $\text{if...then...else...}$

Jeder Name kann nur einmal als Bezeichnung gewählt werden und somit Bezeichnung nur eines einzigen Ausdrucks sein. Namen sind ebenfalls Ausdrücke. Ausdrücke und ihre Namen dürfen sich in Ausdrücken wechselseitig vertreten (mit '==' in der Bedeutung von 'ist wertgleich mit'):

$$(x+y) * (\text{if } (y+z)>0 \text{ then } (y+z)*(x+y) \text{ else } (y+z)/2)$$

$$== b * (\text{if } a>0 \text{ then } a*b \text{ else } a/2)$$

$$== b * c$$

# Namensvereinbarungen mit Argumenten

Namensvereinbarungen können **Argumente** erhalten:

```
quadrat n      = n*n
flaeche l b    = l*b
volumen l b h  = l*b*h
antwort_auf_alle_Fragen = 42
```

Namen sind unabhängig von der Zahl ihrer Argumente Ausdrücke. Der **Wert** von Namen mit ***m*** Argumenten ist eine ***m*-stellige Funktion**. Ist ***m*** gleich null, spricht man von einer **Konstante** oder einer **0-stelligen Funktion**, was eine einheitliche Sicht von Ausdrücken als Ausdrücke von **funktionalem Wert** (oder kürzer als **funktionale Ausdrücke**) ermöglicht:

```
quadrat  :: Int -> Int           -- 1-stellig
flaeche  :: Int -> Int -> Int    -- 2-stellig
volumen  :: Int -> Int -> Int -> Int -- 3-stellig
antwort_auf_alle_Fragen :: Int   -- 0-stellig
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

1552/17



# Namensvereinbarungen mit Bezug

Namensvereinbarungen können aufeinander Bezug nehmen, auch auf sich selbst. In diesem Fall spricht man von **Rekursion**:

```
fac n      = if n <= 0 then 1 else n * fac (n-1)
binom n k  = div (fac n) (fac k * fac (n-k))
```

Die Werte von `fac` und `binom` sind Funktionen:

```
fac  :: Int -> Int           -- 1-stellig
binom :: Int -> Int -> Int    -- 2-stellig
```

Argumente können **implizit** sein:

```
fib = \n -> if n==0 then 0
          else if n==1 then 1
                else fib (n-2) + fib (n-1)
```

Der Wert von `fib` ist ebenfalls eine Funktion:

```
fib :: Int -> Int           -- 1-stellig
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Aufgebrochene Namensvereinbarungen

Namensvereinbarungen für  $m$ -stellige Funktionen,  $m \geq 1$ , können in **mehrere Gleichungen** aufgebrochen werden, die einer Fallunterscheidung entsprechen:

$$\text{fac}' \ 0 = 1$$

$$\text{fac}' \ n = n * \text{fac}' \ (n-1)$$

$$\text{fib}' \ 0 = 0$$

$$\text{fib}' \ 1 = 1$$

$$\text{fib}' \ n = \text{fib}' \ (n-2) + \text{fib}' \ (n-1)$$

Die Werte von  $\text{fac}'$  und  $\text{fib}'$  sind Funktionen:

$\text{fac}' :: \text{Int} \rightarrow \text{Int}$  -- 1-stellig

$\text{fib}' :: \text{Int} \rightarrow \text{Int}$  -- 1-stellig

# Namensvereinbarungen mit Wächtern

Gleichungen können mit Bedingungen, sog. **Wächtern**, versehen und vor ungewollter Anwendung geschützt werden:

```
fac'' n
```

```
| n == 0 = 1
```

```
| n >= 0 = n * fac'' (n-1)
```

```
fib'' n
```

```
| n == 0 = 0
```

```
| n == 1 = 1
```

```
| n >= 0 = fib'' (n-2) + fib'' (n-1)
```

Mögl. Mehrdeutigkeiten durch Überschneidungen (`fac''`: `n==0` mit `n>=0`; `fib''`: `n==0`, `n==1` mit `n>=0`) werden (in Haskell) durch Prüfung der Wächter **von oben nach unten** aufgelöst.

Die Werte von `fac''` und `fib''` sind Funktionen:

```
fac'' :: Int -> Int
```

```
-- 1-stellig
```

```
fib'' :: Int -> Int
```

```
-- 1-stellig
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

1555/17

# A.3

## Operanden und Werte von Ausdrücken

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Werte von Ausdrücken, Ausdruckswerte

Ausdrücke können einen **elementaren Wert** (Zahlen, Zeichen, Wahrheitswerte,...) oder einen **funktionalen Wert** (Funktion auf den ganzen Zahlen, Funktion auf Zeichenreihen in Wahrheitswerte,...) haben.

- Der Wert des Ausdrucks `a` vereinbart durch:

`a = (17+4)*2`

ist **elementar**, ganzzahlig vom Wert `42`; in Haskell:

`a :: Int`.

- Der Wert der Ausdrucks `fac''' n` vereinbart durch:

`fac''' n = if n==0 then 1 else n * fac''' (n-1)`

ist **funktional**, eine Funktion auf den ganzen Zahlen; in Haskell: `fac''' :: Int -> Int`.

# Operanden von Ausdrücken

Ausdrücke können als Operanden (andere) Ausdrücke

- ▶ elementaren Werts haben:

$\text{fac } (17+4)*2-39$   
Operand elem. Werts, ganzzahlig

$\text{fib } (\text{fac } (17+4)*2-39)$   
Operand elem. Werts, ganzzahlig

- ▶ funktionalen Werts haben:

$\text{map } (\backslash n \rightarrow n+1) [1,2,3]$   
Operand fkt. Werts, Inkrementfunktion

$\text{foldr } (*) 1 [1,2,3]$   
Operand fkt. Werts, Multiplikation

# Werte von Ausdrücken

Werte von Ausdrücken können

► **elementar** sein:

```
fac (17+4)*2-39 ->> 6 :: Int
fib (fac (17+4)*2-39) ->> fib 6 ->> 8 :: Int
map (\n-> n+1) [1,2,3] ->> [2,3,4] :: [Int]
foldr (*) 1 [3,4,5] ->> 60 :: Int
```

► **funktional** sein:

```
map (\n-> n+1) :: [Int] -> [Int]      -- 1-stellig
foldr (*) :: Int -> [Int] -> Int      -- 2-stellig
foldr (*) 1 :: [Int] -> Int          -- 1-stellig
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Funktionen, Funktionen höherer Ordnung

## Ausdrücke

- ▶ eines funktionalen Werts heißen **Funktionen**.
- ▶ mit mindestens einem funktionalwertigen Operanden oder Wert (-teil) heißen **Funktionen höherer Ordnung** (oder **Funktionale**).

**Funktionen höherer Ordnung** sind also **Funktionen**, **Funktionen** mit speziellen Eigenschaften ihrer Argumente oder/und Resultate.

Statt von **Operand(en)** und **Wert** spricht man bei Funktionen auch von **Argument(en)** und **Resultat**.



# Funktionen erster Ordnung

Funktionen, die keine Funktionen höherer Ordnung sind, heißen

► Funktionen erster Ordnung.

Funktionen erster Ordnung sind Funktionen mit elementarwertigen Argumenten und elementarwertigem Resultat.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# A.4

## Funktionen und Polymorphie: Erstrangige Sprachelemente

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Funktionen

...sind in **funktionalen** Sprachen

- ▶ **erstrangige** Sprachelemente (engl. **first-class citizens**):  
Ausdrücke funktionalen Werts dürfen (fast überall) stehen, wo auch Ausdrücke elementaren Werts stehen dürfen und umgekehrt.

...sind in **imperativen** Sprachen

- ▶ **zweitrangige** Sprachelemente (engl. **second-class citizens**):  
Ausdrücke funktionalen Werts oder Prozeduren dürfen nicht überall stehen; in vielen Sprachen noch auf Argumentposition als funktionale oder prozedurale Argumente von Funktionen und Prozeduren, kaum als Resultat von Funktionen.

# Polymorphie

...ist in **funktionalen** Sprachen

- **erstrangiges** Sprachelement (engl. **first-class citizen**):

`map :: (a -> b) -> [a] -> [b]`

`foldr :: (a -> b -> b) -> b -> [a] -> b`

`flip :: (a -> b -> c) -> (b -> a -> c)`

`length :: [a] -> Int`

Der Typ jeder Funktion ist grundsätzlich von den konkreten Typen der Argumentwerte so weit entkoppelt wie irgend möglich.

...ist in **imperativen** Sprachen

- **zweitrangiges** Sprachelement (engl. **second-class citizen**):

Möglichkeiten zur polymorphen Typspezifikation fehlen oft völlig oder sind im Vergleich zu funktionalen Sprachen wesentlich limitiert.

# A.5

## Imperative vs. funktionale Programme

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Imperative Programme

...sind eine **Menge von Anweisungsvereinbarungen**, kurz Anweisungen, zusammen mit einer **Anordnung**, in welcher Reihenfolge diese Anweisungen auszuführen sind.

Anweisungen werden in **Prozeduren** zusammengefasst, um das Programm zu strukturieren; Prozeduren werden in **Modulen** zusammengefasst, um das Programm weiter zu strukturieren.

**Anm.:** In objektorientierten Sprachen ist als Bezeichnung **Methode** statt **Prozedur** üblich.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Funktionale Programme

...sind eine Menge von Ausdrucksvereinbarungen, kurz Ausdrücken.

Eine explizite, ausdrückliche Anordnung, in welcher Reihenfolge sie ausgewertet werden sollen, gibt es nicht (Ausnahme: In übersetzten Programmen ist ein Ausdruck durch seinen Namen (in Haskell `main`) als zuerst auszuwertender 'Hauptausdruck' ausgezeichnet).

Einzig Wertabhängigkeiten zwischen Ausdrücken legen lose eine Reihenfolge fest, wenn Ausdrücke zur Auswertung ausgewählt werden.

Ausdrücke werden in Modulen zusammengefasst, um Programme zu strukturieren; eine Zusammenfassung von Ausdrücken zu 'Ausdruckssammlungen' analog einer Prozedur (oder Methode) gibt es nicht; eine Funktion ist ein benannter Ausdruck, keine Ausdruckssammlung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## A.6

# Wertzuweisung vs. Wertvereinbarung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Wertzuweisungen vs. Wertvereinbarungen

Betrachte die Bedeutung von:

$a := (17+4)*2$  (oft  $a = (17+4)*2$  als Schreibw.)

als imperative Wertzuweisung und von:

$b = (17+4)*2$

als funktionale Wertvereinbarung.

# Imp. Wertzuweisung, fkt. Wertvereinbarung

...ähnliche Optik, fundamental andere Bedeutung.

- **Imperativ:** Wertzuweisung, temporär, abänderbar

`a := (17+4)*2`

**Wertzuweisung** für **Speicherzelle a** auf Zeit; Wertzuweisungen sind **temporäre** Wertfestlegungen für **benannte Speicherzellen**, Festlegungen auf Zeit, bis zum nächsten Überschreiben (**abänderbar**, engl. **mutable**).

- **Funktional:** Wertvereinbarung, dauerhaft, unabänderbar

`b = (17+4)*2`

**Wertvereinbarung** für **Ausdrucksnamen b** für die gesamte Programmzukunft; Wertvereinbarungen sind **permanente** Wertfestlegungen für **Ausdrucksnamen**, Festlegungen für immer, auf alle Zeit (**unabänderbar**, engl. **immutable**).

# Imperativ: Wertzuweisungen

## Wertzuweisung:

`a := (17+4)*2`

## Bedeutung:

- ▶ In der mit dem Namen `a` bezeichneten Speicherzelle sei ab jetzt bis auf weiteres (**abänderbar!**) der Wert `42` gespeichert.
- ▶ Diese Festlegung kann in der Zukunft jederzeit und beliebig oft widerrufen und durch eine neue Wertfestlegung ersetzt werden, ebenfalls auf Zeit.
- ▶ Ein Austausch von `a` durch `42` oder umgekehrt hat deshalb (über die Vertauschung hinaus) i.a. einen von außen beobachtbaren Effekt und ist deshalb nicht möglich: Nicht überall, wo `42` stehen darf, darf ohne Bedeutungsunterschied auch `a` stehen oder umgekehrt.

# Funktional: Wertvereinbarungen

## Wertvereinbarung:

$$b = (17+4)*2$$

## Bedeutung:

- ▶ Der Name **b** hat ab jetzt für den gesamten restlichen Programmablauf (**unabänderbar!**) den Wert **42**.
- ▶ Wo immer **42** stehen darf, darf bedeutungsgleich auch **b** stehen und umgekehrt (cum grano salis - mit einem Körnchen Salz (Plinius der Ältere):  $42 = b$  ist nicht möglich, weil 42 kein gültiger Name ist).
- ▶ Ein Austausch von **b** durch **42** oder umgekehrt hat (über die Vertauschung hinaus) keinen von außen beobachtbaren Effekt.

# Imperativ: Variablennamen

...Namen sind **Variablennamen**.

**a** := (17+4)\*2 **Wertzuweisung, a Variablenname**

- ▶ Es besteht auf dem Niveau des Programms eine logische und adressierbare Verknüpfung zwischen dem Variablennamen **a** und einer Speicherzelle, die den Wert der Variablen aufnimmt.
- ▶ Aufgrund der Verknüpfung kann diese Speicherzelle vom Programmierer vom Programm aus angesprochen, gelesen und auch geändert werden.

# Funktional: Ausdrucksnamen

...Namen sind **Ausdrucksnamen**.

**b** = (17+4)\*2 **Wertvereinbarung, b Ausdrucksname**

- Eine logische oder/und adressierbare Verknüpfung des Ausdrucksnamens **b** mit einer Speicherzelle besteht auf dem Niveau des Programms nicht.

Auf der Maschine ist der mit dem Namen **b** verbundene Wert 42 natürlich in einer Speicherzelle gespeichert (wie auch und wo auch sonst, da wir für funktionale und imperative Programme dieselben Maschinen, dieselben Hardware-Architekturen benutzen). Diese Speicherzelle kann allerdings vom Programmierer vom Programm aus weder angesprochen noch gelesen oder gar geändert werden.

## A.7

# Selbstbezügliche Wertzuweisungen, Wertvereinbarungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Sebstbezügliche Wertzuw. und Wertvereinb.

...lassen den konzeptuell fundamentalen Unterschied zwischen

► **imperativen Wertzuweisungen** ( $a := a + 1$ )

und

► **funktionalen Wertvereinbarungen** ( $b = b + 1$ )

besonders hervortreten und auf den Punkt zu bringen:

Welchen Wert erhält

►  $a$  durch die **imperative Wertzuweisung** ' $a := a + 1$ '?

►  $b$  durch die **funktionale Wertvereinbarung** ' $b = b + 1$ '?  
Kann oder sollte  $b$  durch diese Vereinbarung einen wohldefinierten Wert erhalten? Welchen?



# Imperativ: Selbstbezug in Wertzuweisungen

## Selbstbezügliche Wertzuweisung:

►  $a := a + 1$

## Operationelle Bedeutung:

- Lies den Wert, der in der mit dem Variablennamen **a** verknüpften Speicherzelle abgelegt ist, erhöhe ihn um **1** und überschreibe den gelesenen alten Wert mit dem erhöhten neuen Wert.

## Das heißt:

- Alles ist wohldefiniert! Der alte Wert, der neue Wert und der Weg vom alten zum neuen Wert. Die Ausführung der selbstbezüglichen Wertzuweisung **terminiert!**
- Es gibt ein Konzept von vorher und nachher, von Wert vor und nach der Ausführung, von **Zustand!**
- Der **neue** Wert von **a** ist der **alte** Wert von **a** erhöht um **1**.

# Funktional: Selbstbezug in Wertvereinbarungen

## Selbstbezügliche Wertvereinbarung:

►  $b = b + 1$

## Bedeutung:

- Ausdrucksname  $b$  und rechtsseitiger Ausdruck  $b+1$  haben denselben Wert; der Wert von  $b$  ist Lösung der Gleichung  $b = b + 1$ .

## Das heißt:

- Der Wert von  $b$  ist gleich dem Wert von  $b+1$ , welcher gleich dem Wert von  $(b+1)+1$  ist, welcher gleich dem Wert von  $((b+1)+1)+1$  ist, welcher gleich dem Wert von  $((b+1)+1)+1$  ist usw: Ein **unendlicher Regress!**
- Die operationelle Berechnung des Wert von  $b$  über die Berechnung des Werts von  $b+1$ :
- $b \rightarrow b+1 \rightarrow (b+1)+1 \rightarrow ((b+1)+1)+1 \rightarrow \dots$
- terminiert nicht! Der Wert von  $b$  ist **undefiniert!**

# Funktional: Selbstbezug in Wertvereinbarungen

## Wichtig:

- ▶ Funktional ist weder die Rede von einem alten noch von einem neuen Wert von  $b$ , sondern nur vom Wert von  $b$ .
- ▶ Funktional gibt es **kein Konzept** von vorher und nachher, kein Konzept eines Wertes von vor oder nach der Auswertung eines Ausdrucks, kein Konzept eines Zustands!
- ▶ Die Gleichung  $b = b + 1$  hat **keine Lösung!**
- ▶ Der Wert von  $b$  als Lösung der Gleichung  $b = b + 1$  existiert deshalb nicht in einem wohldefinierten Sinn, er ist undefiniert.
- ▶ Im operationellen Versuch, den Wert einer selbstbezüglichen Wertvereinbarung zu berechnen, zeigt sich das daran, dass die Berechnungsfolge **nicht terminiert!**

$b \rightarrow b+1 \rightarrow (b+1)+1 \rightarrow ((b+1)+1)+1 \rightarrow \dots$

# Auf den Punkt gebracht:

Die **imperative Wertzuweisung** ' $a := a + 1$ ' bedeutet:

- ▶ Erhöhe den in Speicherzelle **a** befindlichen Wert um **1**.

Die **funktionale Wertvereinbarung** ' $b = b + 1$ ' bedeutet:

- ▶ Finde eine Lösung für die Gleichung  $b = b + 1$ .

Der **Bedeutungsunterschied** ist **offensichtlich** und er ist **fundamental**.

# A.8

## Problem- und Lösungssicht: Imperativ vs. funktional

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Problemsicht

...für **imperative** und **funktionale** Programmierung **ident**, gegeben und beschrieben durch:

- ▶ Menge  $\mathcal{P}$  von Problemen, Menge  $\mathcal{L}$  von Lösungen.
- ▶ Probleminstanzen  $p \in \mathcal{P}$  mit Lösungen  $l_p \in \mathcal{L}$ .

Beispiel:

- ▶  $\mathcal{P}$ : Menge der Listen ganzer Zahlen.
- ▶  $\mathcal{L}$ : Menge der aufsteigend sortierten Listen ganzer Zahlen.
- ▶  $p$ : Die Zahlenliste 42, 4, 4711, 17.
- ▶  $l_p$ : Die aufsteigend sortierte Zahlenliste 4, 17, 42, 4711.
- ▶ **Gesucht**: Sortierverfahren (z.B. **Quicksort**) für Listen ganzer Zahlen.

# Imperative Lösungssicht

## Ansatz:

- Schreibe ein **imperatives Programm**  $\pi$  über der Variablenmenge  $V$  zur Lösung von Problem instanzen  $p \in \mathcal{P}$ .

(Wichtig: Eine Variablenbelegung von  $\pi$  ist eine Abb.  $\nu : V \rightarrow W$ , die jeder Variablen aus  $V$  einen Wert aus einem Wertebereich  $W$  zuordnet.)

## Korrektheitsannahme (partielle Korrektheit):

- Wenn eine **initiale** Variablenbelegung  $\nu_{init}$  von  $\pi$  eine Problem instanzen  $p \in \mathcal{P}$  beschreibt und  $\pi$  angesetzt auf  $\nu_{init}$  mit der **finalen** Variablenbelegung  $\nu_{final}$  terminiert, dann beschreibt  $\nu_{final}$  eine Lösung  $l_p \in \mathcal{L}$  von  $p$ .

# Frage, Aufforderung an ein imperatives Prg.

Ist  $\pi$  ein **imperatives Programm**,  $p \in \mathcal{P}$  eine Probleminstance und  $\nu_{init}$  eine  $p$  beschreibende **initiale Variablenbelegung** von  $\pi$ , so lauten **Frage** und **Aufforderung** an  $\pi$ :

**Frage:** Welches ist die

- ▶ **finale Variablenbelegung** (und damit Beschreibung einer Lösung  $l_p \in \mathcal{L}$  von  $p$ )

wenn  $\pi$  auf  $\nu_{init}$  angesetzt wird, d.h. wenn  $\pi$  mit der durch  $\nu_{init}$  gegebenen Variablenbelegung gestartet wird?

**Aufforderung:**

- ▶ Führe deine **Instruktionen** beginnend mit den Werten der initialen Variablenbelegung aus und liefere die Werte der **finalen Variablenbelegung**!



# Funktionale Lösungssicht

## Ansatz:

- Schreibe ein funktionales Programm  $\phi$  über der Namensmenge  $N$  zur Lösung von Probleminstanzen  $p \in \mathcal{P}$ .

(Wichtig: Ein Ausdruck  $\alpha$  ist ein Ausdruck über der Namensmenge  $N$  von  $\phi$ , wenn  $\alpha$  induktiv ausschließlich aus Grundoperanden, Grundoperationen und Namen aus  $N$  aufgebaut ist.)

## Korrektheitsannahme (partielle Korrektheit):

- Wenn ein Ausdruck  $\alpha$  über der Namensmenge  $N$  von  $\phi$  eine Probleminstanz  $p \in \mathcal{P}$  beschreibt und die Auswertung von  $\alpha$  durch  $\phi$  mit dem Wert  $w$  terminiert, dann beschreibt  $w$  eine Lösung  $l_p \in \mathcal{L}$  von  $p$ .

# Frage, Aufforderung an ein funktionales Prg.

Ist  $\phi$  ein **funktionales Programm** (also ein System von Gleichungen),  $p \in \mathcal{P}$  eine Probleminstance und  $\alpha$  ein  $p$  beschreibender **Ausdruck** über der Namensmenge von  $\phi$ , so lauten **Frage** und **Aufforderung** an  $\phi$ :

**Frage:** Welches ist der

- Wert von  $\alpha$  (und damit die Beschreibung einer Lösung  $l_p \in \mathcal{L}$  von  $p$ )

unter den Gleichungen von  $\phi$ , d.h. was ist der Wert von  $\alpha$ , wenn die in  $\phi$  gegebenen Gleichungen gelten?

**Aufforderung:**

- Liefere den **Wert** von Ausdruck  $\alpha$  unter Lösung der Gleichungen deines Gleichungssystems!

# Zum Vergleich

Aufforderung an ein **imperatives** Programm:

- ▶ Führe deine **Instruktionen** beginnend mit durch  $\nu_{init}$  gegebenen Variablenbelegung aus und liefere die **finale Variablenbelegung**  $\nu_{final}$ !

Aufforderung an ein **funktionales** Programm:

- ▶ Liefere den **Wert** von Ausdruck  $\alpha$  unter Lösung der Gleichungen deines Gleichungssystems!

Der **Unterschied** ist **offensichtlich** und **fundamental**:

- ▶ Eine Aufgabe **imperativ** zu lösen erfordert deshalb eine andere **Herangehens-** und **Denkweise** als **funktional** und **umgekehrt**!
  - ▶ **Imperativ**: Denken in **Instruktionen** und ihren **Effekten**.
  - ▶ **Funktional**: Denken in **Gleichung(ssystem)en** und **Eigenschaften** ihrer **Lösungen**.

# Aus dieser unterschiedlichen Sicht

...auf die Welt und ihre Probleme ergeben sich Unterschiede  
im Denken, Tun und Produkt

- ▶ imperativer
- ▶ funktionaler

Programmierung.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Denken, Tun, Produkt imp. Programmierung

## Denken:

- ▶ Denken in Instruktionen und ihren Effekten.

## Tun:

Spezifiziere Instruktionen an den Rechner:

- ▶ Tu dies, tu das, tu jenes,...

mit dem Ziel, ihn in die Lage zu versetzen, durch Ausführung der vorgegebenen Instruktionen eine initiale Variablenbelegung in eine

- ▶ finale Variablenbelegung zu überführen.

## Produkt:

Ein imperatives Programm als Gesamtheit seiner Instruktionen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Denken, Tun, Produkt fkt. Programmierung

## Denken:

- ▶ Denken in Gleichung(ssystem)en und Eigenschaften ihrer Lösungen.

## Tun:

Spezifiziere Gleichungen für den Rechner:

- ▶ Diese, jene, folgende,...

mit dem Ziel, ihn in die Lage zu versetzen, durch Lösung der vorgegebenen Gleichungen einen Ausdruck in seinen

- ▶ Wert zu überführen.

## Produkt:

Ein funktionales Programm als Gesamtheit seiner Gleichungen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Das Schreiben funktionaler Programme

...ist deshalb das Schreiben von Gleichung(ssystem)en.

Gleichungen funktionaler Programme beschreiben fast alle

- ▶ polymorphe echte Funktionen (d.h. 1- oder mehrst. Fkt.):

```
map :: (a -> b) -> [a] -> [b]
map f xs = if xs==[] then []
           else f (head xs) : map f (tail xs)
```

Die allermeisten beschreiben:

- ▶ polymorphe Funktionen höherer Ordnung:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

- ▶ einige polymorphe Funktionen erster Ordnung:

```
length :: [a] -> Int
length xs = if xs==[] then 0 else 1 + length (tail xs)
```

- ▶ wenige unechte Funktionen (d.h. 0-stell. Fkt., Konstanten):

```
pi :: Float
pi = 3.14
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe A.8.1 – Logisierung

Wie sehen die Welt und ihre Probleme durch die Brille **logischer Programmierung** aus? Wie sehen im Sinn dieses Abschnitts die

- ▶ **logische** Lösungssicht (**Ansatz**, **Korrektheitsannahme**)
- ▶ **Frage**, **Aufforderung** an ein **logisches** Programm

aus?

Wodurch sind **Herangehens-** und **Denkweise** gekennzeichnet, ein Problem durch **logische Programmierung**, z.B. ein **Prolog-**Programm, zu lösen?

- ▶ **Logisch Programmieren** heißt: **Denken** in ...?

Was folgt daraus für **Tun** und **Produkt** im Fall **logischer Programmierung**?



## A.9

Welcher Problemlösungstyp bin ich?

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Imperative Problemlösungssicht

## Konzeptionell:

- **Denken** in Instruktionen und ihren Effekten.

## Operationell:

- Haarklein jeden Schritt bis hin zum allerletzten Detail unzweideutig anweisen.

Ich erklär's nur einmal – Sortieren geht so:

```
quickSort (L,low,high)
  if low < high
    then splitInd = partition (L,low,high)
    quickSort (L,low,splitInd-1)
    quickSort (L,splitInd+1,high) fi

partition (L,low,high)
  l = L[low]
  left = low
  for i = low+1 to high do
    if L[i] <= l then left = left+1
    swap (L[i],L[left]) fi od
  swap (L[low],L[left])
  return left
```

Abtreten zum Sortieren! Im Laufschrift. Marsch!



In der Gefahr besteht die Schwierigkeit nie darin,  
Menschen zu finden, die gehorchen werden,  
sondern Männer, die befehlen können.

George Bernard Shaw (1854-1900)  
irischer Schriftsteller

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

# Funktionale Problemlösungssicht

## Konzeptionell:

- Denken in Gleichung(ssystem)en und Eigenschaften ihrer Lösungen.

## Operationell:

- Präzise die Eigenschaften der Lösung beschreiben.

Lieber Sesselkreis, liebe Sesselkreisler,  
ich wünsche mir, dass meine Liste permutiert eine  
der folgenden zwei Eigenschaften erfüllt:

```
(1) quickSort []      = []  
(2) quickSort (n:ns) = quickSort [m | m <- ns, m <= n]  
                        ++ [n]  
                        ++ quickSort [m | m <- ns, m > n]
```

...den lästigen Rest zur Wunschverwirklichung übernehmen  
ihre Sesselkreisler, begeistert, zwanglos, sofort.

Der ideale Mensch fühlt Freude,  
wenn er anderen einen Dienst erweisen kann.

Aristoteles (384 - 322 v.Chr.)  
griech. Philosoph



Man soll den Menschen nie sagen,  
wie sie etwas tun sollen,  
sondern nur, was sie tun sollen.  
Dann wird ihr Einfallsreichtum einen verblüffen.

George S. Patton (1885-1945)  
amerik. General

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Imperativer und fkt. Lösungsweg im Vergleich

## Imperativ:

Aufgabe des inspirierenden **Instruktors**:

Exaktes Anweisen des **einzuschlagenden Lösungswegs**, des 'wie' zur Lösung gelangen.



## Funktional:

Aufgabe des eleganten **Sesselkreis-Delegators**:

Präzises Beschreiben der **essentiellen Eigenschaften der Lösung**, des 'was' der Lösung, unter weitreichender Freistellung des konkreten Wegs zur Lösung für den Sesselkreis bei freilich determiniertem Ergebnis!



# Instrukteur, Delegateur: Erforderliches Wissen

## Imperativ:

- Wie sieht die Lösung aus? Wie komme ich zur Lösung hin?

Bsp.: Der Instrukteur muss wissen: Wenn auf meine Liste folgende Schritte pippifein angewendet werden, ist sie sortiert.

```
quickSort (L,low,high)
  if low < high
    then splitInd = partition (L,low,high)
         quickSort (L,low,splitInd-1)
         quickSort (L,splitInd+1,high) fi

partition (L,low,high)
  l = L[low]
  left = low
  for i = low+1 to high do
    if L[i] <= l then left = left+1
    swap (L[i],L[left]) fi od
  swap (L[low],L[left])
  return left
```

## Funktional:

- Was erwarte ich von der Lösung?

Bsp.: Der Delegateur erwartet: Meine Liste ist sortiert, wenn sie Gleichung (1) oder Gleichung (2) erfüllt (und denkt sich: Zu wissen, was ich will, war durchaus genug mit Arbeit verbunden; Wegschrittfolgen auch noch wissen zu sollen, wäre zu viel erwartet!).

```
(1) quickSort [] = []
(2) quickSort (n:ns) = quickSort [m | m <- ns, m <= n]
    ++ [n] ++ quickSort [m | m <- ns, m > n]
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Imp. vs. fkt. Lsg.-Weg: Aufgabenlastverteilung

Auf wem liegt **konzeptionell** und **operationell** die größere Last?

- ▶ Dem **handlungsfixierten imperativ** vorgehenden **Instrukteur**?



- ▶ Dem **ergebnisorientierten funktio-**  
**nal** vorgehenden **Delegateur**?



# Übungsaufgabe A.9.1 – Schönheitssehnsucht

Instrukteur und imperativer Programmierer erfreuen sich an der exakten Beschreibung der Schönheiten des Wegs zur Lösung, Delegateur und funktionaler Programmierer an der präzisen Beschreibung der Schönheiten der Lösung selbst.

Welche Parallelen gibt es zum Zitat von Antoine de Saint-Exupéry? Welche Doppelrolle kommt den 'Männern' im Zitat in dieser Sicht zu?

Wenn du ein Schiff bauen willst,  
dann trommle nicht Männer zusammen,  
um Holz zu beschaffen, Aufgaben zu vergeben  
und die Arbeit einzuteilen, sondern lehre die Männer  
die Sehnsucht nach dem weiten unendlichen Meer.

Antoine de Saint-Exupéry (1900-1944)  
franz. Schriftsteller

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Imperative oder funktionale Programmierung

...eine Frage (auch?! ) von Typ, Persönlichkeit, Führungsstil:

- Sie bevorzugen Ordnung, klare Hierarchien und Ansagen und wissen ohnehin am besten, wie eine Aufgabe zu erledigen ist?

**Imperative** Programmierung ist Ihr Ding. Probieren Sie nichts anderes.



- Sie bevorzugen ein kooperatives, harmonisches Arbeitsumfeld, in dem partizipativ, konsensual mit schließlicher Zufriedenstellung aller ohne der Rede werten eigenen Beitrags Ihr ganz persönliches Wunschergebnis erarbeitet wird?

**Funktionale** Programmierung könnte Ihr Ding sein. Probieren Sie es!

**Gleichungen** bilden Ihr **kooperatives** und **harmonisches Arbeitsumfeld**!





# Typauswertung und Prognose

- (1) Sie sind als **Instrukteur** geboren auf die Welt gekommen ohne sonderliche Veranlassung sich zu verändern? Vor Ihnen
  - ▶ liegt eine **harte Zeit**; Sie werden sich glücklich schätzen, anschließend in Ihre **Welt der klaren Ansagen** zurückkehren zu dürfen.
- (2) Sie haben sich als **Instrukteur** nie wirklich, nie rundum wohlfühlt? Als **Delegateur** werden Sie möglicherweise
  - ▶ **aufblühen** und darin ihre Bestimmung finden; Ihre **Rückkehr** in die Welt der klaren Ansagen ist **zweifelhaft**.
- (3) Sie können situationselastisch gleichermaßen gut als **Instrukteur** wie als **Delegateur** agieren? Sie werden in beiden
  - ▶ **Welten**, der **imperativen** und der **funktionalen** Programmierung, **reüssieren**; das wünsche ich Ihnen!

# Ihr Typ ist nicht dabei?

...instruieren scheint Ihnen zu harsch, delegieren zu ausnutzend, logisieren zu kalt, empathie- und herzlos?

Der vierte Weg: Augenhöhe, Wertschätzung!

Du, Rechner — ich glaube, Deine Ergebnisse — stimmen nicht — was denkst Du? — Wir müssen mal — reden.

Hallo!? — Rechner? — Hallooo? — Wir müssen — reeden!! — Hörst Du nicht? — ICH WILL MIT DIR REEEDEN...

Wenn die Alternativen unattraktiv erscheinen, sicher einen Versuch wert. Oder, Blaise Pascals Logik-Variante, fünfter Weg:

Logik des Herzens.

Blaise Pascal (1623-1662)

franz. Philosoph, Mathematiker und Physiker

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Oder 6. Weg: Den Rechner nehmen 'wie er ist'!

- ▶ **Imperativ** zu programmieren verlangt als **Instrukteur** zu **denken** und zu **handeln**.



Gib keine Befehle, die man nicht vollbringen kann.

Äsop (6. Jh. v.Chr.)  
griech. Fabeldichter

- ▶ **Funktional** zu programmieren verlangt als **Delegateur** zu **denken** und zu **handeln**.



Delegiere, was andere besser können.

sprichwörtl., lebensklug

- ▶ **Logisch** zu programmieren verlangt als **Spockteur** zu **denken** und zu **handeln**.



Logik, die Anatomie des Denkens.

John Locke (1632-1704)  
engl. Philosoph

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Das heißt: Ist mein Programmierumfeld

...hineingeworfen oder selbstgewählt (LVA, Firma, mein Projekt), in dem ich erfolgreich sein will, eine Welt

- klarer Ansagen, so denke ich beim Programmieren imperativ in Instruktionen und ihren Effekten und handle als Instrukteur.



- harmonischen und konsensualen Ausgleichs, so denke ich beim Programmieren funktional in Gleichungssystemen und Eigenschaften ihrer Lösungen und handle als Delegateur.



- faszinierender Empathie- und Herzlosigkeit, so denke ich beim Programmieren logisch in Formeln und ihrer logischen Konsequenzen und handle als Spockteur.



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Bedenkenswertes zu(m) Denken und Handeln

Das Handeln ist eine Folge des Denkens.

Franz Peter Künzel (\* 1925)  
dt. Lektor und Redakteur

Denken ist oft schwerer als man denkt.

Werner Mitsch (1936-2009)  
dt. Aphoristiker

Denken ist die schwerste Arbeit, die es gibt.  
Das ist wahrscheinlich der Grund, warum  
sich so wenige damit beschäftigen.

Henry Ford (1863-1947)  
amerikan. Unternehmer

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

Es gibt keinen Ausweg,  
den ein Mensch nicht beschreitet,  
um die tatsächliche Arbeit  
des Denkens zu vermeiden.

Thomas A. Edison (1847-1931)  
amerikan. Erfinder und Unternehmer

Verzicht auf Denken ist geistige Bankrotterklärung.

Albert Schweitzer (1875-1965)  
elsäss. evang. Theologe und Arzt

Denken hilft. Meistens.  
Nachdenken noch mehr.  
Am meisten vorher.

Hochschullehrerweisheit, unbekannt (leider)

Auch im Denken gibt es eine Zeit des Pflügens  
und eine Zeit der Ernte.

Ludwig Wittgenstein (1889-1951)  
österreich. Philosoph

Denken lernt man nicht an Regeln zum Denken,  
sondern an Stoff zum Denken.

Jean Paul (1763-1825)  
dt. Schriftsteller

Worüber wir nicht ernsthaft nachgedacht haben,  
vergessen wir schnell.

Marcel Proust (1871-1922)  
franz. Schriftsteller

Denken ist nicht dasselbe wie Gelesenhaben.

Antonio Machado (1875-1939)  
span. Lyriker

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

Eine Sache lernt man, indem man sie macht.

Cesare Pavese (1908-1950)  
italien. Schriftsteller

Für das Können gibt es nur einen Beweis, das Tun.

Marie von Ebner-Eschenbach (1830-1916)  
österreich. Schriftstellerin

Wer immer tut, was er schon kann,  
bleibt immer das, was er schon ist.

Henry Ford (1863-1947)  
amerikan. Unternehmer

Der Weg zum Ziel beginnt an dem Tag,  
an dem du die hundertprozentige Verantwortung  
für dein Tun übernimmst.

Dante (um 1265-1321)  
italien. Schriftsteller

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Übungsaufgabe A.9.2 – Feldstudie

Beobachten Sie Ihre Umgebung. Hat Aristoteles recht?

Einen guten und einen weniger guten **imperativen Programmierer** unterscheidet:

Niemand kann gut befehlen,  
der nicht zuvor gehorchen gelernt hat.

Aristoteles (384 - 322 v.Chr.)  
griech. Philosoph

Ein **funktionaler Programmierer** folgt dem Prinzip:

Der Gebildete treibt die Genauigkeit nicht weiter,  
als es der Natur der Sache entspricht.

Aristoteles (384 - 322 v.Chr.)  
griech. Philosoph

Cogita, omne simile claudicat.  
Bedenke, jedes Gleichnis hinkt.  
lat., sprichwörtl.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe A.9.3 – Syllogilei (1)

Die **Syllogistik** ist die Lehre von den **Syllogismen**, den logischen **Schlüssen**, bei denen **deduktiv** vom **Allgemeinen** auf das **Besondere** geschlossen wird. Die **Syllogistik** geht auf **Aristoteles** zurück, dargestellt in den beiden *Analytiken*.

Berühmtes Beispiel eines **Syllogismus** ist:

Alle Menschen sind sterblich.

Sokrates ist ein Mensch.

Also ist Sokrates sterblich.

## Übungsaufgabe A.9.3 – Syllogilei (2)

Ambros Bierce (1842 - Weihnachten/Neujahr 1913/14), amerik. Schriftsteller, Autor von *'The Cynic's Word Book'*, hat in seiner Erklärung der Logik den arithmetischen Syllogismus als Schluss eingeführt. Mr. Spock, ist (syl)logisch alles o.k.?

Logik, die: Die Kunst des Denkens und Argumentierens in strenger Übereinstimmung mit den Beschränkungen und Unfähigkeiten des menschlichen Missverstehens. Die Grundlage der Logik ist der Syllogismus, der aus einem Obersatz, einem Untersatz und einer Konklusion besteht – in etwa so:

**Obersatz:** Sechzig Männer können eine Arbeit sechzigmal so schnell vollbringen wie ein einziger.

**Untersatz:** Ein Mann kann ein Pfostenloch in 60 Sekunden graben, also...

**Konklusion:** Sechzig Männer können ein Pfostenloch in 1 Sekunde graben.

Dies kann man einen arithmetischen Syllogismus nennen, durch den wir aufgrund der Verbindung von Logik und Mathematik eine doppelte Gewissheit erlangen und daher gleich zwiefach gesegnet sind (Übersetzung von Dr. Michael Siefener).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# A.10

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang A (1)

-  Neal Ford. *Functional Thinking: Why Functional Programming is on the Rise*. IBM developerWorks, 11 pages, 2013. [www.ibm.com/developerworks/java/library/j-ft20/j-ft20-pdf.pdf](http://www.ibm.com/developerworks/java/library/j-ft20/j-ft20-pdf.pdf)
-  Konrad Hinsén. *The Promises of Functional Programming*. Computing in Science and Engineering 11(4):86-90, 2009.
-  John Hughes. *Why Functional Programming Matters*. The Computer Journal 32(2):98-107, 1989.
-  Konstantin Läufer, George K. Thiruvathukal. *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*. Computing in Science and Engineering 11(5):68-75, 2009.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10



Kap. 11

Teil V

Kap. 12

Kap. 13/17

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang A (2)

-  Mihai Maruseac. *Haskell: A Language for Modern Times*. Crossroads, the ACM Magazine for Students 24(1):64-66, 2017.
-  Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.
-  Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.
-  Michael Siefener. *Aus dem Wörterbuch des Teufels*. marixverlag, 2011.
-  Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# B

## Formale Rechenmodelle

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# B.1

## Turing-Maschinen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Turing-Maschine

## Definition B.1.1 (Turing-Maschine)

Eine **Turing-Maschine**  $TM$  ist ein 'schwarzer' Kasten, der über einen **Lese/Schreibkopf** mit einem **unendlichen Rechenband** verbunden ist.

Das Rechenband ist in einzelne (nummerierte) **Felder** eingeteilt, von denen zu jeder Zeit genau eines unter dem Lese/Schreibkopf liegt.

$TM$  kann **interne Zustände**  $0, 1, 2, 3, \dots$  aus  $\mathbb{N}_0$  annehmen; der interne Zustand  $0$  ist der **Startzustand** von  $TM$ .

Es gibt eine Möglichkeit,  $TM$  einzuschalten; das Arbeiten und Abschalten von  $TM$  erfolgt selbsttätig.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Turing-Tafel, Turing-Programm

## Definition B.1.2 (Turing-Tafel, Turing-Programm)

Eine **Turing-Tafel** (oder **Turing-Programm**)  $T$  über einem (endlichen) Zeichenvorrat  $\mathcal{A}$  ist eine Tafel mit 4 Spalten und  $m + 1$  Zeilen,  $m \geq 0$ :

$i_0$	$a_0$	$b_0$	$j_0$
$i_1$	$a_1$	$b_1$	$j_1$
$\dots$			
$i_k$	$a_k$	$b_k$	$j_k$
$\dots$			
$i_m$	$a_m$	$b_m$	$j_m$

Dabei gilt für alle  $0 \leq k \leq m$ :

- ▶  $i_k, j_k \in \mathbb{N}_0$ .
- ▶  $a_k \in \mathcal{A} \cup \{\mathfrak{b}\}$  (d.h.  $\mathfrak{b} \notin \mathcal{A}$ ).
- ▶  $b_k \in \mathcal{A} \cup \{\mathfrak{b}\} \cup \{L, R\}$  (d.h.  $L, R \notin \mathcal{A} \cup \{\mathfrak{b}\}$ ).
- ▶ Für alle Paare  $(i_p, a_p) \in \mathbb{N}_0 \times (\mathcal{A} \cup \{\mathfrak{b}\})$  gilt:  $(i_p, a_p)$  kommt höchstens einmal als Zeilenanfang vor.

# Turing-Tafel, Turing-Programm (figs.)

Die Elemente jeder Zeile von  $T$  bedeuten für eine  $T$  beobachtende Turing-Maschine  $TM$ :

- ▶ Das **erste** Element einen **Zustand**, den  $TM$  mit ihrem **internen Zustand** zu vergleichen hat.
- ▶ Das **zweite** Element ein **Zeichen** aus  $\mathcal{A} \cup \{\mathfrak{b}\}$ , das  $TM$  mit dem aktuell unter ihrem **Lese/Schreibkopf** liegenden Zeichen zu vergleichen hat.
- ▶ Das **dritte** Element den Befehl für  $TM$ : 'Drucke  $b_k$ ', falls  $b_k \in \mathcal{A} \cup \{\mathfrak{b}\}$ ; den Befehl 'Gehe nach links', falls  $b_k = L$ ; den Befehl 'Gehe nach rechts', falls  $b_k = R$  (d.h. positioniere den Lese/Schreibkopf je ein Feld weiter links bzw. rechts).
- ▶ Das **vierte** Element den **internen Folgezustand**, den  $TM$  nach Ausführung des Befehls annimmt.

# Arbeitsweise einer Turing-Maschine

## Definition B.1.3 (Arbeitsweise einer TM)

Sei  $TM$  eine Turing-Maschine,  $T$  eine endliche Turing-Tafel:

- ▶ Bei Einschalten nimmt  $TM$  den internen Zustand  $0$  an und der Lese/Schreibkopf positioniert sich über Feld  $0$  des Rechenbands; dabei wird angenommen, dass auf jedem Feld des Rechenbands etwas steht.
- ▶ Einschaltet beobachtet  $TM$  die endliche Turing-Tafel (oder Turing-Programm)  $T$  und kann abhängig davon:
  - ▶ Den Lese/Schreibkopf ein Feld nach links oder nach rechts bewegen.
  - ▶ Zeichen  $a_1, a_2, \dots, a_n$  eines (endlichen) Zeichenvorrats  $\mathcal{A}$  sowie das Sonderzeichen  $\flat \notin \mathcal{A}$  auf Felder des Rechenbands drucken;  $\flat$  steht dabei für das Leerzeichen. Jedes Drucken löscht und überschreibt das vorher auf dem Feld befindliche Zeichen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Arbeitsweise einer Turing-Maschine (figs.)

## Arbeiten/rechnen:

- ▶ Stimmen interner Zustand von **TM** und aktuelles Zeichen unter dem Lese/Schreibkopf von **TM** mit dem Anfang einer Zeile in **T** überein, so führt **TM** den Befehl dieser (eindeutig bestimmten) Zeile aus und nimmt anschließend den in dieser Zeile genannten Folgezustand als neuen internen Zustand an.

## Abschalten/terminieren:

- ▶ **TM** schaltet sich ab, wenn interner Zustand von **TM** und aktuelles Zeichen unter dem Lese/Schreibkopf von **TM** nicht als Anfang einer Zeile in **T** vorkommen.

# Turing-berechenbar

## Definition B.1.4 (Turing-berechenbar)

Eine partiell definierte Funktion  $f$  ist **Turing-berechenbar** gdw. es gibt eine Turing-Tafel  $T$ , eine  $T$  beobachtende Turing-Maschine **TM**, eine eindeutige Codierung der Argument- und Bildwerte von  $f$  als Rechenbandinhalte, so dass **TM** angesetzt auf die Codierung eines Arguments  $a$  von  $f$  mit der Codierung des Bildwerts  $b$  von  $a$  auf dem Rechenband terminiert, wenn  $f(a) = b$  definiert ist, oder nicht terminiert oder mit einem speziellen Fehlerwert als Bandinhalt terminiert, wenn  $f(a)$  nicht definiert ist.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Beispiele zweier Turing-Programme (1)

...gegeben durch die Turing-Tafeln  $T_1$  und  $T_2$  über dem elementigen Zeichenvorrat  $\{\mid\}$ .

1. Turing-Tafel  $T_1$ :

0	b		1
1		L	2

Eine  $T_1$  beobachtende Turing-Maschine  $TM$  terminiert angesetzt auf das **Rechenband** (vgl. **Übungsaufgabe B.1.4**):

...	b	b	b			...		b	b	...
	-2	-1	0	1	2		n	n+1	n+2	

im Zustand 2 mit dem **Bandinhalt**:

...	b	b				...		b	b	...
	-2	-1	0	1	2		n	n+1	n+2	

# Beispiele zweier Turing-Programme (2)

## 2. Turing-Tafel $T_2$ :

0	b	R	1
1	b	b	8
1		b	2
2	b	R	3
3		R	3
3	b	R	4
4		R	4
4	b		5
5		R	5
5	b		6
6		L	6
6	b	L	7
7		L	7
7	b	R	1

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Beispiele zweier Turing-Programme (3)

Eine  $T_2$  beobachtende Turing-Maschine **TM** terminiert ange-  
setzt auf das **Rechenband** (vgl. Übungsaufgabe B.1.4):

...	␣	␣			...		␣	␣	...
	-1	0	1	2		n	n+1	n+2	

mit dem **Bandinhalt**:

...	␣	␣	␣	␣	...	␣	␣			...		␣	␣	...
	-1	0	1	2		n	n+1	n+2	n+3		3n+1	3n+2		

# Übungsaufgabe B.1.4

Eine Turing-Maschine  $TM$ , die angesetzt auf ein Band mit je einem Strich auf den Feldern  $1$  bis  $n$ , das ansonsten leer ist, die **Turing-Tafel**

- ▶  $T_1$  beobachtet, fügt zu den vorhandenen Strichen einen Strich im Feld  $0$  hinzu, geht einen Schritt nach links und terminiert im Zustand  $2$ .
  - ▶  $T_2$  beobachtet, löscht sukzessive alle Striche auf den Feldern  $1$  bis  $n$  und druckt  $2n$  Striche neu auf das Band beginnend im Feld  $n+2$  und endend im Feld  $3n+1$ .
1. Überprüfe dieses Verhalten von  $TM$  durch schrittweises händisches Nachvollziehen der Arbeitsschritte von  $TM$  für jeweils folgenden **Anfangsinhalt** des **Rechenbands**:

...	⌞	⌞	⌞				⌞	⌞	...
	-2	-1	0	1	2	3	4	5	

2. In welchem Zustand terminiert  $TM$   $T_2$  beobachtend?

# B.2

## Markov-Algorithmen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Markov-Tafel

## Definition B.2.1 (Markov-Tafel)

Eine **Markov-Tafel**  $T$  über einem (endlichen) Zeichenvorrat  $\mathcal{A}$  ist eine Tafel mit 5 Spalten und  $m + 1$  Zeilen,  $m \geq 0$ :

0	$a_0$	$i_0$	$b_0$	$j_0$
1	$a_1$	$i_1$	$b_1$	$j_1$
...				
$k$	$a_k$	$i_k$	$b_k$	$j_k$
...				
$m$	$a_m$	$i_m$	$b_m$	$j_m$

Dabei gilt:

- ▶  $k, i_p, j_p \in \mathbb{N}_0$ ,  $0 \leq k \leq m$ .
- ▶  $a_p, b_p \in \mathcal{A}^*$  mit  $\mathcal{A}^*$  Menge der Worte über  $\mathcal{A}$ ;  $\varepsilon$  bezeichnet das leere Wort aus  $\mathcal{A}^*$ .

# Markov-Algorithmus

## Definition B.2.2 (Markov-Algorithmus)

Ein **Markov-Algorithmus**

$$M = (T, Z, E, A, f_M)$$

ist gegeben durch

1. Eine **Zwischenkonfigurationsmenge**  $Z = \mathcal{A}^* \times \mathbb{N}_0$ .
2. Eine **Eingabekonfigurationsmenge**  $E \subseteq \mathcal{A}^* \times \{0\}$ .
3. Eine **Ausgabekonfigurationsmenge**  $A \subseteq \mathcal{A}^* \times [m + 1.. \infty)$ .
4. Eine **Markov-Tafel**  $T$  über dem (endlichen) Zeichenvorrat  $\mathcal{A}$  mit  $m + 1$  Zeilen mit der durch  $T$  definierten (partiellen) **Überföhrungsfunktion**

$$f_M : Z \rightarrow Z$$

# Markov-Algorithmus (fgs.)

...wobei die Überföhrungsfunktion

$$f_M : Z \rightarrow Z$$

definiert ist durch:

$$\forall x \in \mathcal{A}^*. \forall k \in \mathbb{N}_0.$$

$$f_M(x, k) =_{df} \begin{cases} (x, i_k) & \text{falls } k \leq m \text{ und } a_k \text{ keine Teil-} \\ & \text{zeichenreihe von } x \text{ ist.} \\ (\bar{x}b_k\bar{\bar{x}}, j_k) & \text{falls } k \leq m \text{ und } x = \bar{x}a_k\bar{\bar{x}}, \text{ wobei} \\ & \text{die Lnge von } \bar{x} \text{ minimal ist.} \\ \text{undefiniert} & \text{falls } k > m. \end{cases}$$

Der Markov-Alogorithmus  $M$  terminiert, wenn das Resultat der Überföhrungsfunktion undefiniert ist.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

## Definition B.2.3 (Markov-berechenbar)

Eine partiell definierte Funktion  $f$  ist **Markov-berechenbar** gdw. es gibt einen Markov-Algorithmus  $M$ , eine eindeutige Codierung der Argument- und Bildwerte von  $f$  über dem Zeichenvorrat von  $M$ , so dass  $M$  angesetzt auf die Codierung eines Arguments  $a$  von  $f$  mit der Codierung des Bildwerts  $b$  von  $a$  terminiert, wenn  $f(a) = b$  definiert ist, oder nicht terminiert oder mit einer speziellen Zeichenreihe als Fehlerwert terminiert, wenn  $f(a)$  nicht definiert ist.

# Beispiele dreier Markov-Algorithmen (1)

...in Form der Markov-Tafeln  $T_1$ ,  $T_2$  und  $T_3$ .

1. Markov-Tafel  $T_1$  über dem Alphabet  $\{|\}$ :

$$0 \quad \varepsilon \quad i_0 \quad | \quad 1$$

2. Markov-Tafel  $T_2$  über dem Alphabet  $\{|\mathfrak{b}\}$ :

$$0 \quad \mathfrak{b} \quad i_0 \quad \varepsilon \quad 1$$

wobei in  $T_1$  und  $T_2$  der Wert  $i_0$  (verschieden von null) beliebig aus  $\mathbb{N}_1$  gewählt werden darf.



# Beispiele dreier Markov-Algorithmen (2)

## 3. Markov-Tafel $T_3$ über dem Alphabet $\{ |, \alpha, \beta, \mathfrak{b} \}$ :

0	$\mathfrak{b}$	$i_0$	$\mathfrak{b}\beta$	1
1	$\beta $	2	$ \beta$	1
2	$ \mathfrak{b}$	7	$\mathfrak{b}$	3
3	$\mathfrak{b}$	$i_3$	$\mathfrak{b}\alpha$	4
4	$\alpha $	6	$ \alpha$	5
5	$\beta$	$i_5$	$\beta $	4
6	$\alpha$	$i_6$	$\varepsilon$	2
7	$\mathfrak{b} $	8	$\mathfrak{b}$	7
8	$\mathfrak{b}\beta$	$i_8$	$\varepsilon$	9

wobei die Werte  $i_0, i_3, i_5, i_6, i_8$  (verschieden von null) beliebig aus  $\mathbb{N}_1$  gewählt werden dürfen.

# Beispiele dreier Markov-Algorithmen (3)

Stellt man die natürlichen Zahlen als Strichfolgen im Einer-system dar (die leere Strichfolge  $\varepsilon$  repräsentiert die null die Strichfolge aus  $n$  Strichen die Zahl  $n$ ), so gilt: Die durch die Markov-Tafel

- ▶  $T_1$  induzierte Überföhrungsfunktion realisiert die Nachfolger-Funktion einer als Strichfolge gegebenen natürlichen Zahl: ' $nf \quad ||| = ||||$ '
- ▶  $T_2$  induzierte Überföhrungsfunktion realisiert die Addition zweier als Strichfolgen gegebener natürlicher Zahlen: ' $|| + ||| = ||||$ '
- ▶  $T_3$  induzierte Überföhrungsfunktion realisiert die Multiplikation zweier als Strichfolgen gegebener natürlicher Zahlen: ' $|| * ||| = |||||$ '

# Übungsaufgabe B.2.4

Überprüfe anhand einiger selbstgewählter Beispiele durch schrittweises händisches Nachvollziehen der Arbeitsschritte der durch die Markov-Tafeln  $T_1$ ,  $T_2$  und  $T_3$  induzierten Überföhrungsfunktionen, dass die zugehörigen **Markov-Algorithmen**

- ▶  $M_1$  angewendet auf eine als Strichfolge gegebene natürliche Zahl deren **Nachfolger** dargestellt als Strichfolge berechnet.
- ▶  $M_2$  angewendet auf zwei als Strichfolgen gegebene durch genau ein Leerzeichen **b** voneinander getrennte natürliche Zahlen deren **Summe** dargestellt als Strichfolge berechnet.
- ▶  $M_3$  angewendet auf zwei als Strichfolgen gegebene durch genau ein Leerzeichen **b** voneinander getrennte natürliche Zahlen deren **Produkt** dargestellt als Strichfolge berechnet.

## B.3

# Primitiv rekursive Funktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Primitiv rekursive Funktionen

## Definition B.3.1 (Primitiv rekursive Funktionen)

Eine Funktion  $f$  heißt **primitiv rekursiv**, wenn  $f$  ausgehend von den Grundfunktionen  $\lambda x.0$  und  $\lambda x.x + 1$  durch endlich viele Anwendungen expliziter Transformationen, Kompositionen und primitiver Rekursionen hervorgeht.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Transformation, Komposition

## Definition B.3.2 (Explizite Transformation)

Eine Funktion  $g$  geht aus einer Funktion  $f$  durch eine explizite Transformation hervor, wenn es  $e_1, \dots, e_n$  gibt, so dass jedes  $e_i$  entweder eine Konstante aus  $\mathbb{IN}$  oder eine Variable  $x_i$  ist, so dass für alle  $\bar{x}^m \in \mathbb{IN}^m$  gilt:

$$g(x_1, \dots, x_m) = f(e_1, \dots, e_n)$$

## Definition B.3.3 (Komposition)

Sind  $f : \mathbb{IN}^k \rightarrow \mathbb{IN}_\perp$ ,  $g_i : \mathbb{IN}^n \rightarrow \mathbb{IN}_\perp$  für  $i = 1, \dots, k$  Funktionen, dann geht  $h : \mathbb{IN}^k \rightarrow \mathbb{IN}_\perp$  durch Komposition aus den Funktionen  $f, g_1, \dots, g_k$  hervor gdw. für alle  $\bar{x}^n \in \mathbb{IN}^n$  gilt:

$$h(\bar{x}^n) = \begin{cases} f(g_1(\bar{x}^n), \dots, g_k(\bar{x}^n)) & \text{falls jedes } g_i(\bar{x}^n) \neq \perp \text{ ist} \\ \perp & \text{sonst} \end{cases}$$

# Primitive Rekursion

## Definition B.3.4 (Primitive Rekursion)

Sind  $f : \mathbb{N}^n \rightarrow \mathbb{N}_\perp$  und  $g : \mathbb{N}^{n+2} \rightarrow \mathbb{N}_\perp$  Funktionen, dann geht  $h : \mathbb{N}^{n+1} \rightarrow \mathbb{N}_\perp$  durch **primitive Rekursion** aus  $f$  und  $g$  hervor gdw. für alle  $\bar{x}^n \in \mathbb{N}^n, t \in \mathbb{N}$  gilt:

$$h(0, \bar{x}^n) = f(\bar{x}^n)$$

$$h(t+1, \bar{x}^n) = \begin{cases} g(t, h(t, \bar{x}^n), \bar{x}^n) & \text{falls } h(t, \bar{x}^n) \neq \perp \\ \perp & \text{sonst} \end{cases}$$

# B.4

## $\mu$ -rekursive Funktionen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# $\mu$ -rekursive Funktionen, Minimierung

## Definition B.4.1 ( $\mu$ -rekursive Funktionen)

Eine Funktion  $f$  heit  $\mu$ -rekursiv, wenn  $f$  ausgehend von den Grundfunktionen  $\lambda x.0$  und  $\lambda x.x + 1$  durch endlich viele Anwendungen expliziter Transformationen, Kompositionen, primitiver Rekursionen und Minimierungen totaler Funktionen hervorgeht.

## Definition B.4.2 (Minimierung)

Ist  $g : \mathbb{N}^{n+1} \rightarrow \mathbb{N}_\perp$  eine Funktion, dann geht  $h : \mathbb{N}^n \rightarrow \mathbb{N}_\perp$  aus  $g$  durch Minimierung hervor gdw. fr alle  $\bar{x}^n \in \mathbb{N}^n$  gilt:

$$h(\bar{x}^n) = \begin{cases} t & \text{falls } t \in \mathbb{N} \text{ die kleinste Zahl ist mit } g(t, \bar{x}^n) = 0 \\ \perp & \text{sonst} \end{cases}$$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# B.5

## Leseempfehlungen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10






Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (1)

-  Friedrich L. Bauer. *Historische Notizen – Wer erfand den von-Neumann-Rechner?* Informatik-Spektrum 21(3):84-89, 1998.
-  Cristian S. Calude. *People and Ideas in Theoretical Computer Science*. Springer-V., 1999.
-  Luca Cardelli. *Global Computation*. ACM SIGPLAN Notices 32(1):66-68, 1997.
-  Gregory J. Chaitin. *The Limits of Mathematics*. Journal of Universal Computer Science 2(5):270-305, 1996.
-  Gregory J. Chaitin. *The Limits of Mathematics – A Course on Information Theory and the Limits of Formal Reasoning*. Springer-V., 1998.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (2)

-  Gregory J. Chaitin. *The Unknowable*. Springer-V., 1999.
-  Paul Cockshott, Greg Michaelson. *Are There New Models of Computation? Reply to Wegner and Eberbach*. The Computer Journal 50(2):232-247, 2007.
-  S. Barry Cooper, Benedikt Löwe, Andrea Sorbi (Hrsg). *New Computational Paradigms: Changing Conceptions of What is Computable*. Springer-V., 2008.
-  B. Jack Copeland. *The Broad Conception of Computation*. American Behavioral Scientist 40(6):690-716, 1997.
-  B. Jack Copeland. *The Church-Turing Thesis*. The Stanford Encyclopedia of Philosophy, 2002.  
<http://plato.stanford.edu/entries/church-turing>

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10


Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (3)

-  B. Jack Copeland. *Accelerating Turing Machines*. Minds and Machines 12(2):281-301, 2002.
-  B. Jack Copeland. *Hypercomputation*. Minds and Machines 12(4):461-502, 2002.
-  B. Jack Copeland, Eli Dresner, Diane Proudfoot, Oron Shagrir. *Viewpoint: Time to Reinspect the Foundations? Questioning if Computer Science is Outgrowing its Traditional Foundations*. Communications of the ACM 59(11):34-36, 2016.
-  B. Jack Copeland, Carl J. Posy, Oron Shagrir (Hrsg.). *Computability: Turing, Gödel, Church, and Beyond*. MIT Press, 2013.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (4)



B. Jack Copeland, Oron Shagrir. *The Church-Turing Thesis: Logical Limit or Breachable Barrier?* Communications of the ACM 62(1):66-74, 2019.



Martin Davis. *What is a Computation?* Kapitel in Lynn A. Steen (Hrsg.), *Mathematics Today – Twelve Informal Essays*. Springer-V., 241-268, 1978.



Martin Davis. *Mathematical Logic and the Origin of Modern Computers*. Studies in the History of Mathematics, Mathematical Association of America, 137-165, 1987.  
Nachdruck in: Rolf Herken (Hrsg.), *The Universal Turing Machine – A Half-Century Survey*, Kemmerer&Unverzagt und Oxford University Press, 149-174, 1988.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10




Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (5)

-  Martin Davis. *The Universal Computer: The Road from Leibniz to Turing*. W.W. Norton and Company, 2000.
-  Martin Davis. *The Myth of Hypercomputation*. Christof Teuscher (Hrsg.), Alan Turing: Life and Legacy of a Great Thinker, Springer-V., 195-212, 2004.
-  Martin Davis. *The Church-Turing Thesis: Consensus and Opposition*. In Proceedings of the 2nd Conference on Computability in Europe – Logical Approaches to Computational Barriers (CiE 2006), Springer-V., LNCS 3988, 125-132, 2006.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (6)



Martin Davis. *Why There is No Such Discipline as Hypercomputation*. Applied Mathematics and Computation 178(1):4-7, Special issue on Hypercomputation, 2006.



John W. Dawson Jr. *Gödel and the Origin of Computer Science*. In Proceedings of the 2nd Conference on Computability in Europe – Logical Approaches to Computational Barriers (CiE 2006), Springer-V., LNCS 3988, 133-136, 2006.



Peter J. Denning. *The Field of Programmers Myth*. Communications of the ACM 47(7):15-20, 2004.



Peter J. Denning, Peter Wegner. *Introduction to What is Computation*. The Computer Journal 55(7):803-804, 2012.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11




Teil V

Kap. 12

Kap. 13



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (7)

-  Jon Doyle. *What is Church's Thesis? An Outline*. Minds and Machines 12(4):519-520, 2002.
-  Charles E.M. Dunlop. Book review on: M. Gams, M. Paprzycki, X. Wu (Hrsg.). *Mind Versus Computer: Were Dreyfus and Winograd Right?*, Frontiers in Artificial Intelligence and Applications Vol. 43, IOS Press, 1997. Minds and Machines 10(2):289-296, 2000.
-  Eugene Eberbach, Dina Q. Goldin, Peter Wegner. *Turing's Ideas and Models of Computation*. Christof Teuscher (Hrsg.), Alan Turing: Life and Legacy of a Great Thinker, Springer-V., 159-194, 2004.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (8)



Bertil Ekdahl. *Interactive Computing does not Supersede Church's Thesis*. In Proceedings of the 17th International Conference on Computer Science, Association of Management and the International Association of Management, Vol. 17, No. 2, Part B, 261-265, 1999.



Matjaž Gams. *The Turing Machine may not be the Universal Machine – A Reply to Dunlop*. Minds and Machines 12(1):137-142, 2002.



Matjaž Gams. *Alan Turing, Turing Machines and Stronger*. Informatica 37(1):9-14, 2013.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10




Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (9)

-  Dina Q. Goldin, Scott B. Smolka, Paul C. Attie, Elaine L. Sonderegger. *Turing Machines, Transition Systems, and Interaction*. Information and Computation Journal 194(2):101-128, 2004.
-  Dina Q. Goldin, Peter Wegner. *The Interactive Nature of Computing: Refuting the Strong Church-Rosser Thesis*. Minds and Machines 18(1):17-38, 2008.
-  Yuri Gurevich. *What is an Algorithm?* In Proceedings of SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current Trends in the Theory and Practice of Computer Science. Springer-V., LNCS 7147, 31-42, 2012.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (10)

-  Yuri Gurevich. *Unconstrained Church-Turing Thesis cannot Possibly be True*. CoRR abs/1901.04911, 2019.  
<https://arxiv.org/abs/1901.04911>
-  Brosl Hasslacher. *Beyond the Turing Machine*. In Rolf Herken (Hrsg.), *The Universal Turing Machine: A Half-Century Survey*. Springer-V., 2. Auflage, 387-402, 1995.
-  Neil D. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press, 1997.
-  Saul B. Kripke. *The Church-Turing “Thesis” as a Special Corollary of Gödel’s Completeness Theorem*. In B. Jack Copeland, Carl J. Posy, Oron Shagrir (Hrsg.) *Computability: Turing, Gödel, Church, and Beyond*. MIT Press, 77-104, 2013.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (11)

-  John MacCormick. *What can be Computed? A Practical Guide to the Theory of Computation*. Princeton University Press, 2018.
-  Emil Leon Post. *Finite Combinatory Processes: Formulation I*. The Journal of Symbolic Logic 1(3):103-105, 1936.
-  Michael Prasse, Peter Rittgen. *Bemerkungen zu Peter Wegners Ausführungen über Interaktion und Berechenbarkeit*. Informatik-Spektrum 21(3):141-146, 1998.
-  Michael Prasse, Peter Rittgen. *Why Church's Thesis Still Holds. Some Notes on Peter Wegner's Tracts on Interaction and Computability*. The Computer Journal 41(6):357-362, 1998.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10





Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (12)

-  Edna E. Reiter, Clayton M. Johnson. *Limits of Computation: An Introduction to the Undecidable and the Intractable*. Chapman and Hall, 2012.
-  Uwe Schöning. *Complexity Theory and Interaction*. In Rolf Herken (Hrsg.), *The Universal Turing Machine – A Half-Century Survey*. 2. Auflage, Springer-V., 519-536, 1995.
-  Jack T. Schwartz. *Do the Integers Exist? The Unknowability of Arithmetic Consistency*. *Communications on Pure and Applied Mathematics* 58:1280-1286, 2005.
-  Dennis Shasha. *Future of Computing: Inspiration from Nature*. *Crossroads, the ACM Magazine for Students* 18(3):38-39, 2012.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10




Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (13)

-  Wilfried Sieg. *Church without Dogma: Axioms for Computability*. In S. Barry Cooper, Benedikt Löwe, Andrea Sorbi (Hrsg.), *New Computational Paradigms - Changing Conceptions of What is Computable*, Springer-V., 139-152, 2008.
-  Alan Turing. *On Computable Numbers, with an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society* 42(2):230-265, 1936.  
Correction, *ibid*, 43:544-546, 1937.
-  Alan Turing. *Computing Machinery and Intelligence*. *Mind* 59:433-460, 1950.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (14)



Jan van Leeuwen, Jirí Wiedermann. *On Algorithms and Interaction*. In Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS 2000), Springer-V., LNCS 1893, 99-112, 2000.



Jan van Leeuwen, Jirí Wiedermann. *The Turing Machine Paradigm in Contemporary Computing*. In Björn Enquist, Wilfried Schmidt (Hrsg.), *Mathematics Unlimited – 2001 and Beyond*. Springer-V., 1139-1155, 2001.



Jan van Leeuwen, Jirí Wiedermann. *Beyond the Turing Limit: Evolving Interactive Systems*. In Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2001), Springer-V., LNCS 2234, 90-109, 2001.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11






Teil V

Kap. 12

Kap. 13



# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (15)

-  Robin Milner. *Elements of Interaction: Turing Award Lecture*. Communications of the ACM 36(1):78-89, 1993.
-  Hava T. Siegelmann. *Neural Networks and Analog Computation: Beyond the Turing Limit*. Birkhäuser, 1999.
-  Peter Wegner. *Why Interaction is More Powerful Than Algorithms*. Communications of the ACM 40(5):81-91, 1997.
-  Peter Wegner. *Interactive Foundations of Computing*. Theoretical Computer Science 192(2):315-351, 1998.
-  Peter Wegner. *Observability and Empirical Computation*. The Monist 82(1), Issue on the Philosophy of Computation, 1999.  
[www.cs.brown.edu/people/pw/papers/monist.ps](http://www.cs.brown.edu/people/pw/papers/monist.ps)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (16)

 Peter Wegner. *The Evolution of Computation*. The Computer Journal 55(7):811-813, 2012.

 Peter Wegner, Eugene Eberbach. *New Models of Computation*. The Computer Journal 47(1):4-9, 2004.

 Peter Wegner, Dina Q. Goldin. *Interaction, Computability, and Church's Thesis*. Accepted to the British Computer Journal.

[www.cs.brown.edu/people/pw/papers/bcj1.pdf](http://www.cs.brown.edu/people/pw/papers/bcj1.pdf)

 Peter Wegner, Dina Q. Goldin. *Computation Beyond Turing Machines*. Communications of the ACM 46(4):100-102, 2003.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Leseempfehlungen zum vertiefenden und weiterführenden Selbststudium für Anhang B (17)



Peter Wegner, Dina Q. Goldin. *The Church-Turing Thesis: Breaking the Myth*. In Proceedings of the 1st Conference on Computability in Europe – New Computational Paradigms (CiE 2005), Springer-V., LNCS 3526, 152-168, 2005.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

C

Andere funktionale Sprachen

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Schlaglichter

...auf **ausgewählte** andere **funktionale Programmiersprachen** und wesentliche ihrer **Eigenschaften**:

- ▶ **ML**: Starker Wettbewerber von **Haskell** mit **sofortiger** (engl. **eager**) Auswertung.
- ▶ **Lisp**: Der ***Oldtimer*** unter den funktionalen Sprachen.
- ▶ **APL**: Ein sprachlicher **Exot**.
- ▶ ...

# C.1

## ML

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# ML: Eine Sprache mit 'sofortiger' Auswertung

ML, eine **strikte** funktionale Sprache.

Wichtige **Eigenschaften**:

- ▶ Starke Typisierung mit Typinferenz, keine Typklassen.
- ▶ Umfangreiches Typkonzept für Module und abstrakte Datentypen (ADTs).
- ▶ Lexical scoping, curryfizieren (wie Haskell).
- ▶ Zahlreiche Erweiterungen (z.B. in **OCaml**) auch für imperative und objektorientierte Programmierung.
- ▶ Sehr gute theoretische Fundierung.

# ML-Programmbeispiel: Module/ADTs in ML

```
structure S = struct
  type 't Stack          = 't list;
  val  create            = Stack nil;
  fun  push x (Stack xs) = Stack (x::xs);
  fun  pop (Stack nil)   = Stack nil;
      | pop (Stack (x::xs)) = Stack xs;
  fun  top (Stack nil)   = nil;
      | top (Stack (x::xs)) = x;
end;

signature st = sig type q; val push: 't -> q -> q; end;

structure S1:st = S;
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# C.2

## Lisp

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Lisp: Der 'Oldtimer' fkt. Programmierspr.

Lisp, eine bewährte und weiterhin häufig verwendete strikte funktionale Sprache mit imperativen Zusätzen.

Wichtige Eigenschaften:

- ▶ Einfache, interpretierte Sprache, dynamisch typisiert.
- ▶ Listen sind gleichzeitig Daten und Funktionsanwendungen.
- ▶ Nur lesbar, wenn Programme gut strukturiert sind.
- ▶ Erfolgreicher Einsatz in vielen Bereichen, insbesondere künstliche Intelligenz, Expertensysteme.
- ▶ Umfangreiche Bibliotheken, leicht erweiterbar.
- ▶ Sehr gut zur Metaprogrammierung geeignet

# Ausdrücke in Lisp

Beispiele für Symbole:    A            (Atom)  
                              austria    (Atom)  
                              68000    (Zahl)

Beispiele für Listen:    (plus a b)  
                              ((meat chicken) water)  
                              (unc trw synapse ridge hp)  
                              nil bzw. () entsprechen leerer Liste

Eine **Zahl** repräsentiert ihren **Wert** direkt —  
ein **Atom** ist der **Name eines assoziierten Werts**.

(setq x (a b c)) bindet x global an (a b c)

(let ((x a) (y b)) e) bindet x lokal in e an a und y an b

# Funktionen in Lisp

Das erste Element einer Liste wird normalerweise als Funktion interpretiert, anzuwenden auf die restlichen Listenelemente.

(quote a) bzw. 'a liefert Argument a selbst als Ergebnis.

## Beispiele für primitive Funktionen:

(car '(a b c))	->> a	(atom 'a)	->> t
(car 'a)	->> error	(atom '(a))	->> nil
(cdr '(a b c))	->> (b c)	(eq 'a 'a)	->> t
(cdr '(a))	->> nil	(eq 'a 'b)	->> nil
(cons 'a '(b c))	->> (a b c)	(cond ((eq 'x 'y) 'b)	
(cons '(a) '(b))	->> ((a) b)	(t 'c))	->> c

# Funktionsdefinitionen in Lisp

- ▶ `(lambda (x y) (plus x y))` ist Funktion mit zwei Parametern.
- ▶ `((lambda (x y) (plus x y)) 2 3)` wendet diese Funktion auf die Argumente 2 und 3 an und liefert 5 als Resultat.
- ▶ `(define (add (lambda (x y) (plus x y))))` definiert einen globalen Namen `“add”` für die Funktion.
- ▶ `(defun add (x y) (plus x y))` ist abgekürzte Schreibweise dafür.

## Beispiel:

```
(defun reverse (l) (rev nil l))  
(defun rev (out in)  
  (cond ((null in) out)  
        (t (rev (cons (car in) out) (cdr in))))))
```

# Closures in Lisp

- ▶ Kein **curryfizieren** in Lisp, sog. **closures** als Ersatz.
- ▶ **Closures**: lokale Bindungen behalten Wert auch nach Verlassen der Funktion.

Beispiel: 

```
(let ((x 5))  
  (setf (symbol-function 'test)  
        #'(lambda () x)))
```

- ▶ Praktisch: Funktion gibt **closure** zurück.

Beispiel: 

```
(defun create-function (x)  
  (function (lambda (y) (add x y))))
```

- ▶ **Closures** sind flexibel, aber **curryfizieren** ist viel einfacher.

# Dynamisches vs. statisches Binden

...engl. *dynamic scoping*, *static scoping*.

- ▶ Lexikalisch: Bindung ortsabhängig (Quellcode).
- ▶ Dynamisch: Bindung vom Zeitpunkt abhängig.
- ▶ 'Normales' Lisp: Lexikalisches Binden.

Beispiel: 

```
(setq a 100)
(defun test () a)
(let ((a 4)) (test))
```

 $\Rightarrow 100$

- ▶ Dynamisches Binden durch (defvar a) möglich.

Das obige Beispiel liefert damit 4.

- ▶ Code expandiert, nicht als Funktion aufgerufen (wie C).
- ▶ Definition: Erzeugt Code, der danach evaluiert wird.

**Beispiel:**

```
(defmacro get-name (x n)
  (list 'cadr (list 'assoc x n)))
```

- ▶ Expansion und Ausführung:

```
(get-name 'a b) <=> (cadr (assoc 'a b))
```

- ▶ Nur Expansion:

```
(macroexpand '(get-name 'a b)) -> '(cadr (assoc 'a b))
```



# Lisp im Vergleich mit Haskell

Kriterium	Lisp	Haskell
Basis	Einfacher Interpreter	Formale Grundlage
Zielsetzung	Viele Bereiche	Referentiell transparent
Verwendung	Noch häufig	Zunehmend
Sprachumfang	Riesig (kleiner Kern)	Moderat, wachsend
Syntax	Einfach, verwirrend	Modern, Eigenheiten
Interaktivität	Hervorragend	Mit Einschränkungen
Typisierung	Dynamisch, einfach	Statisch, modern
Effizienz	Relativ gut	Relativ gut
Zukunft	Noch lange genutzt	Einflussreich

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# C.3

## APL

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# APL: Ein Exot unter den Sprachen

APL, eine ältere **applikative** (funktionale) Sprache mit **imperativen Zusätzen**.

Wichtige **Eigenschaften**:

- ▶ Dynamische Typisierung.
- ▶ Verwendung speziellen Zeichensatzes.
- ▶ Zahlreiche Funktionen (höherer Ordnung) sind vordefiniert; Sprache aber nicht einfach erweiterbar.
- ▶ Programme sehr kurz und kompakt, aber kaum lesbar.
- ▶ Besonders für Berechnungen mit Feldern gut geeignet.

# APL-Programmentwicklung

...anhand eines **Beispiels**: Berechne d. Primzahlen von 1 bis N:

Schritt 1.  $(\iota N) \circ. | (\iota N)$

Schritt 2.  $0 = (\iota N) \circ. | (\iota N)$

Schritt 3.  $+/[2] 0 = (\iota N) \circ. | (\iota N)$

Schritt 4.  $2 = (+/[2] 0 = (\iota N) \circ. | (\iota N))$

Schritt 5.  $(2 = (+/[2] 0 = (\iota N) \circ. | (\iota N))) / \iota N$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# D

## Datentypdeklarationen in Pascal

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Aufzählungstypen in Pascal

```
TYPE jahreszeiten = (fruehling, sommer, herbst, winter);  
    spielfarbe    = (karo, herz, pik, kreuz);  
    werktage      = (montag, dienstag, mittwoch,  
                    donnerstag, freitag);  
    wochenende    = (samstag, sonntag);
```

## Bemerkung:

- ▶ Gleichheits- und Ordnungsrelationen sind auf Aufzählungstypen automatisch definiert (entspricht deriving (Eq, Ord)), so dass Aufzählungstypwerte verglichen werden können, z.B.  
 $\text{karo} = \text{pik} \rightsquigarrow \text{false}$ ,  $\text{karo} < \text{pik} \rightsquigarrow \text{true}$ ,  $\text{kreuz} \geq \text{herz} \rightsquigarrow \text{true}$ ,  $\text{herz} < > \text{kreuz} \rightsquigarrow \text{true}$ .
- ▶ Die Funktionen succ und pred liefern den Nachfolge- und Vorgängerwert eines Werts, die Funktion ord seine Position in der Aufzählung (entspricht deriving Enum), z.B.  $\text{succ}(\text{herz}) \rightsquigarrow \text{pik}$ ,  $\text{pred}(\text{herz}) \rightsquigarrow \text{karo}$ ,  $\text{succ}(\text{kreuz})$  undef.,  $\text{ord}(\text{karo}) \rightsquigarrow 0$ ,  $\text{ord}(\text{kreuz}) \rightsquigarrow 3$ .

# Produkttypen in Pascal

```
TYPE person      = RECORD
    name: ARRAY [1..50] OF char;
    geschlecht: (maennlich, weiblich);
    alter: 0..150
END;

anschrift = RECORD
    gemeinde: ARRAY [1..50] OF char;
    strasse: ARRAY [1..75] OF char;
    hausnr: integer;
    land: ARRAY [1..100] OF char
END;
```

## Bemerkung:

- ▶ Der Typ von `alter` ist hier als **Ausschnittstyp** ganzer Zahlen definiert. Werte des Typs `0..150` sind die Zahlen von `0` bis `150`.
- ▶ **Bereichsüberschreitungen** zur Laufzeit werden **automatisch überprüft** und führen zum **Programmabbruch**.

# Summentypen in Pascal

```
TYPE index1 = 1..5;
TYPE index2 = 1..100;
TYPE traegermedium = (buch, ebuch, dvd, cd);
TYPE bildSchriftUndTonTraeger =
  RECORD
    CASE
      medium: traegermedium OF
        buch: (autor, titel, verlag: ARRAY [index2] OF char;
              auflage: 1..20; lieferbar: boolean);
        ebuch: (autor, titel, verlag: ARRAY [index2] OF char;
              lizenzBisJahr: integer);
        dvd: (titel, regisseur: ARRAY [index2] OF char;
              hauptdarsteller, sprachen: ARRAY [index1, index2]
                OF char);
        cd: (kuenstler, titel: ARRAY [index2] OF char;
              spieldauer: ARRAY [1..3] OF integer)
    END;
END;
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Mengentypen in Pascal

```
TYPE buchstaben = 'a'..'z';
TYPE zutaten = (mehl, zucker, salz, hefe, eier, essig,
               honig, rosinen, mandeln, joghurt, obst)

TYPE buchstabensuppe = SET OF buchstaben;
TYPE rezept = SET OF zutaten;

VAR vokalsuppe, allerleisuppe: buchstabensuppe;
VAR lebkuchen, nachtisch, verdorben: rezept;

vokalsuppe      := ['a','o','e','u']
                  * ['u','a'..'g'];           (Durchschnitt)
allerleisuppe   := ['a'..'z'] - vokalsuppe;   (Differenz)
lebkuchen       := [mehl..salz,eier,honig..mandeln];
nachtisch       := [joghurt,obst];
verdorben       := lebkuchen + [essig];       (Vereinigung)
```

**Bemerkung:** Mengentypen in Pascal besitzen Eigenschaften und Funktionen, die denen von Listentypen und automatischer Listengenerierung in funktionalen Sprachen ähneln.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# E

## Implementierungsaspekte

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Lineare und repetitive Rekursion

...am Beispiel der Fakultätsfunktion.

Formulierung mittels linearer Rekursion:

Funktional (in Haskell)

Imperativ (in Pseudo-Code)

```
fac :: Int -> Int
fac n
| n == 0 = 1
| True  = n * fac (n-1)
```

```
FUNC fac (n: int): int;
{
  IF n == 0 THEN fac := 1
  ELSE fac := n * fac(n-1) FI }
```

Formulierung mittels repetitiver Rekursion:

Funktional (in Haskell)

Imperativ (in Pseudo-Code)

```
fac :: Int -> Int -> Int
fac n res
| n == 0 = res
| True  = fac (n-1) (n*res)
```

```
FUNC fac (n, res: int): int;
{
  IF n == 0 THEN fac := res
  ELSE fac := fac(n-1, n*res) FI }
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

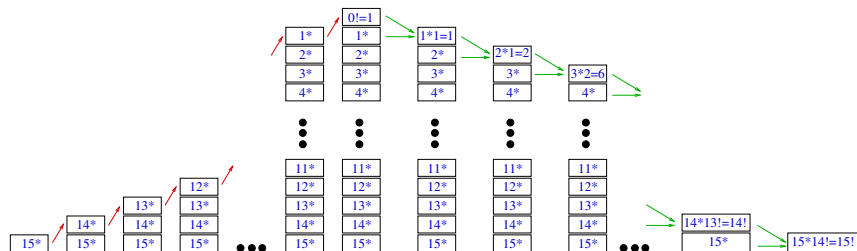
Teil V

Kap. 12

Kap. 13

# Berechnungsablauf

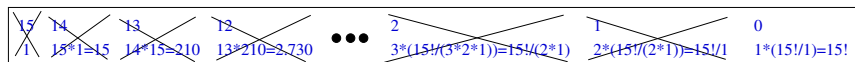
...im Fall linearer Rekursion:



...im Fall repetitiver Rekursion:



Ein (überschreibbarer) Speicherplatz reicht für die Rechnung:



Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

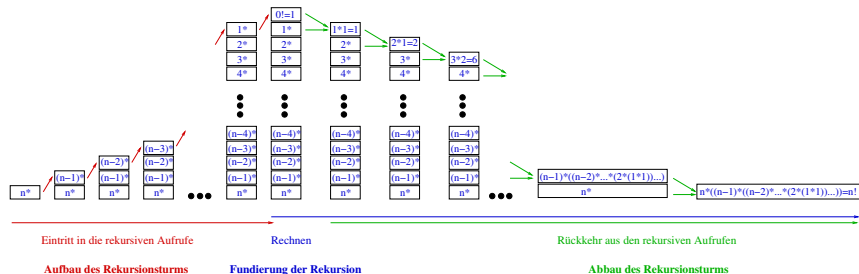
Kap. 11

Teil V

Kap. 12

1684/17

# Laufzeitkeller/-stapel bei linearer Rekursion



- Im Fall **linearer Rekursion** ist der rekursive Aufruf **nicht** die letzte Aktivität in einem Rechenzweig; nach Rückkehr aus dem rekursiven Aufruf wird die Rechnung in der aktuellen Funktionsinkarnation fortgesetzt.
- Für die Fortsetzung der Rechnung sind Werte vorzuhalten, organisiert in Form eines **Laufzeitkellers/-stapels**.
- Auf- und Abbau des Laufzeitstapels erfordern **Verwaltungsaktivitäten**, die über die eigentliche Berechnung hinausgehen- den **zusätzlichen Rechenaufwand zur Laufzeit** verursachen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

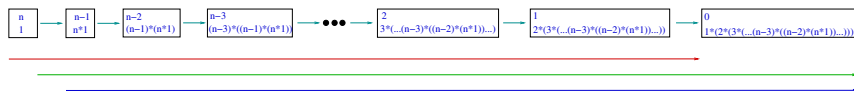
Kap. 11

Teil V

Kap. 12

Kap. 13

# Laufzeitkeller/-stapel bei repetitiver Rekursion



Eintritt, Rechnen und Rückkehr aus rekursiven Aufrufen gehen Hand in Hand.

Werte aus früheren Aufrufen werden nicht benötigt; ein Rekursionsturm entsteht nicht.

- ▶ Im Fall **repetitiver** Rekursion ist der rekursive Aufruf die letzte Aktivität in einem Rechenzweig; sie schließt den aktuellen Aufruf vollständig ab.
- ▶ Eine Rückkehr in diese (oder frühere) Funktionsinkarnationen erfolgt nicht; Werte müssen nicht für eine Fortsetzung der Rechnung vorgehalten werden.
- ▶ Der Laufzeitstapel bleibt **flach**; Rechenaufwand für Auf- und Abbau entsteht nicht.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

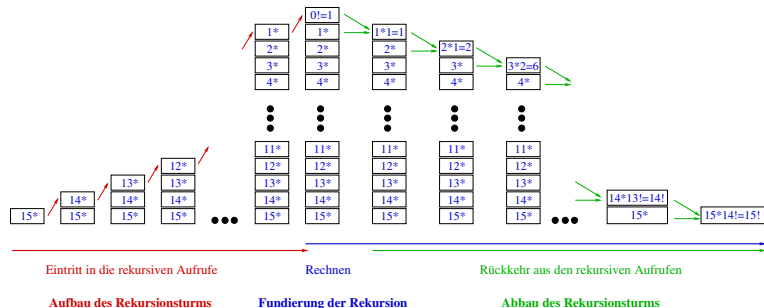
Kap. 10

Kap. 11

Teil V

Kap. 12

# Konkret: Lineare Rekursion am Bsp. von 15!



## Ablauf der Rechnung:

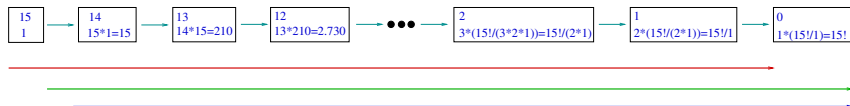
- **Phase 1:** Vollständiger Aufbau des Terms (**Eintritt in die Rekursion**)

$$15 * (14 * (13 * (12 * (\dots * (4 * (3 * (2 * (1 * 1)))))) \dots)))$$

- **Phase 2:** Ausführung der Rechnung (**Rückkehr aus der Rekursion**)

$$\begin{aligned}
 & 15 * (14 * (13 * (12 * (\dots * (4 * (3 * (2 * (1 * 1)))))) \dots))) \\
 \rightsquigarrow & 15 * (14 * (13 * (12 * (\dots * (4 * (3 * (2 * 1)))))) \dots))) \\
 \rightsquigarrow & 15 * (14 * (13 * (12 * (\dots * (4 * (3 * 2)))))) \dots))) \\
 \rightsquigarrow & 15 * (14 * (13 * (12 * (\dots * (4 * 6)))))) \dots))) \\
 \rightsquigarrow & \dots \rightsquigarrow 15! = 1.307.674.368.000
 \end{aligned}$$

# Konkret: Repetitive Rekursion am Bsp. v. 15!



Eintritt, Rechnen und Rückkehr aus rekursiven Aufrufen gehen Hand in Hand.

Werte aus früheren Aufrufen werden nicht benötigt; ein Rekursionsturm entsteht nicht.

## Ablauf der Rechnung:

- 1-phasig: Termaufbau und Rechnung gehen Hand in Hand!

$$\begin{array}{l} 15 / 1 \\ \rightsquigarrow 15 - 1 = 14 / 15 * 1 = 15 \\ \rightsquigarrow 14 - 1 = 13 / 14 * 15 = 210 \\ \rightsquigarrow 13 - 1 = 12 / 13 * 210 = 2.730 \\ \rightsquigarrow \dots \\ \rightsquigarrow 3 - 1 = 2 / 3 * \frac{15!}{3*2*1} = \frac{15!}{2*1} = 653.837.184.000 \\ \rightsquigarrow 2 - 1 = 1 / 2 * \frac{15!}{2*1} = \frac{15!}{1} = 1.307.674.368.000 \\ \rightsquigarrow 1 - 1 = 0 / 1 * \frac{15!}{1} = 15! = 1.307.674.368.000 \end{array}$$

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

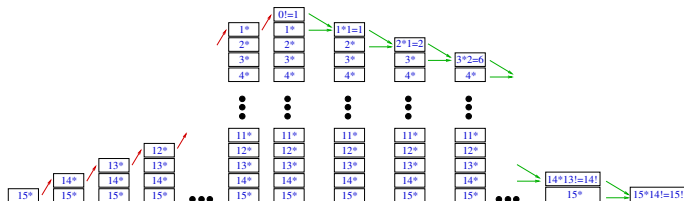
Kap. 13



# Wer wird eher fertig sein?

Jemand, der

- ▶ auf der grünen Wiese für jeden Rechenschnipsel ein neues Stockwerk zu einem Wolkenkratzer in die Höhe türmt, um nach der Ausführung jedes dieser Stockwerke wieder abzureißen, um die Wiese wieder grün zu hinterlassen?



...oder jemand, der

- ▶ die grüne Wiese grün sein lässt, Platz darauf nimmt und die gesamte Rechnung direkt am (Sitz-) Platz vornimmt?

<del>15</del>	<del>14</del>	<del>13</del>	<del>12</del>	...	<del>2</del>	<del>1</del>	<del>0</del>
<del>1</del>	<del>15*1=15</del>	<del>14*15=210</del>	<del>13*210=2,730</del>		<del>3*(15!/(3*2*1))=15!/(2*1)</del>	<del>2*(15!/(2*1))=15!/1</del>	<del>1*(15!/1)=15!</del>

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Mehr dazu

...in [Lehrveranstaltungen zum Übersetzerbau](#), insbesondere:

- ▶ LVA 185.A48 [Übersetzerbau](#)
- ▶ LVA 185.A04 [Optimierende Übersetzer](#)
- ▶ LVA 185.A22 [Seminar aus Übersetzerbau](#)

# Orthogonal

...zu vorigen Überlegungen ist die **Überführung** von

- ▶ repetitiver, linearer Rekursion

in **funktionalen** (oder **imperativen**) Programmen in

- ▶ Iteration (d.h. in **while**-Schleifen).

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Überführung repetitiver Rekursion in Iteration

...am Beispiel der Berechnung des größten gemeins. Teilers.

Formulierung mittels repetitiver Rekursion:

Funktional (in Haskell)

```
ggT :: Int -> Int -> Int
ggT m n
  | n == 0 = m
  | m >= n = ggT (m-n) n
  | True  = ggT (n-m) m
```

Imperativ (in Pseudo-Code)

```
FUNC ggT (m,n: int): int;
{
  IF n=0 THEN ggT := m
  ELSF m >= 0 THEN ggT := ggT(m-n,n)
  ELSE ggT := ggT(n-m,m) FI }
```

Formulierung mittels Iteration (in Pseudo-Code):

```
VAR m,n,ggT: int;
read(m,n);
WHILE not(n==0) DO
  IF m >= 0 THEN m := m-n ELSE (m,n) := (n-m,m) FI OD;
ggT := m.
```

$\hat{=}$  Aufrufen ggT m n, ggT(m,n)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Überführung repetitiver Rekursion in Iteration

...am Beispiel der Fakultätsfunktion.

Formulierung mittels repetitiver Rekursion:

Funktional (in Haskell)

```
fac :: Int -> Int -> Int
fac n res
  | n == 0 = res
  | True   = fac (n-1) (n*res)
```

Imperativ (in Pseudo-Code)

```
FUNC fac (n,res: int): int;
{
  IF n==0 THEN fac := res
  ELSE fac := fac(n-1,n*res) FI }
```

Formulierung mittels Iteration (in Pseudo-Code):

```
VAR n,res,fac: int;
(n,res) := (readint,1);      ≡ Aufrufen fac n 1, fac(n,1)
WHILE not(n==0) DO
  (n,res) := (n-1,n*res) OD;
fac := res.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Überführung linearer Rekursion in Iteration

...am Beispiel der Fakultätsfunktion.

Formulierung mittels linearer Rekursion:

Funktional (in Haskell)

```
fac :: Int -> Int
fac n
  | n == 0 = 1
  | True   = n * fac (n-1)
```

Imperativ (in Pseudo-Code)

```
FUNC fac (n: int): int;
{
  IF n==0 THEN fac := 1
  ELSE fac := n * fac(n-1) FI }
```

Formulierung mittels Iteration (in Pseudo-Code):

```
VAR n, fac: int; ns: [int];
n := readint;
ns := [];
WHILE not(n==0) DO
  (n, ns) := (n-1, n:ns) OD;
ns := 1:ns
WHILE length(ns)>=2 DO
  ns := (fst(ns)*snd(ns)): (tail(tail(ns))) OD;
fac := fst(ns).
```

≡ Aufrufen fac n, fac(n)  
(Phase 1: Sammeln aller Werte)  
(Phase 2: Rechnen)

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Verbesserung: Ersetzen der

...Laufzeitkeller/-stapelsimulation mittels Liste **ns** in Phase 1:

```
VAR n,fac: int; ns: [int];  
n := readint; ≡ Aufrufen fac n, fac(n)  
ns := [];  
WHILE not(n==0) DO (Phase 1: Sammeln aller Werte)  
  (n,ns) := (n-1,n:ns) FI OD;  
ns := 1:ns  
WHILE length(ns)>=2 DO (Phase 2: Rechnen)  
  ns := (fst(ns)*snd(ns)): (tail(tail(ns))) OD;  
fac := fst(ns).
```

durch sofortiges Rechnen mittels einer Variablen **res**:

```
VAR n,fac,res: int;  
n := readint; ≡ Aufrufen fac n, fac(n)  
res := 1;  
WHILE not(n==0) DO  
  (n,res) := (n-1,n*res) OD;  
fac := res.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Beobachtung

Die verbesserte **iterative** Umsetzung **linearer** Rekursion:

```
VAR n,fac,res: int;  
↪ n := readint; ≡ Aufrufen fac n, fac(n)  
↪ res := 1;  
WHILE not(n==0) DO  
  (n,res) := (n-1,n*res) OD;  
fac := res.
```

...ist *de facto* die **iterative** Umsetzung der auf **repetitive** Rekursion zurückgeführten **linearen** Rekursion:

```
VAR n,res,fac: int;  
↪ (n,res) := (readint,1); ≡ Aufrufen fac n 1, fac(n,1)  
WHILE not(n==0) DO  
  (n,res) := (n-1,n*res) OD;  
fac := res.
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Programmanalysen

...erlauben Übersetzern, Rekursionsmuster wie z.B.

- ▶ lineare Rekursion

durch

- ▶ repetitive Rekursion

oder in Abhängigkeit der Zielsprache auch durch

- ▶ Iteration

zu ersetzen.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Exkurs: Turing-Mächtigkeit von WHILE

Die Programmiersprache **W**HILE (der sog. **while-Kern** imperativer Programmiersprachen) mit

$\pi ::= x := a$	(Zuweisung)
if $b$ then $\pi_1$ else $\pi_2$ fi	(Fallunterscheidung)
while $b$ do $\pi_1$ od	(while-Schleife)
$\pi_1; \pi_2$	(Sequentielle Komposition)

wobei

- ▶  $a$  für **Ausdrücke**
- ▶  $b$  für **Wahrheitswertausdrücke**

stehen, ist **Turing-mächtig**.

# Informelle Folgerung

...aus der Turing-Mächtigkeit von **W**HILE:

- ▶ Jedes **rekursive** Programm lässt sich in ein bedeutungs-  
gleiches **W**HILE-Programm überführen, d.h. in ein **itera-**  
**tives** Programm, das ausschließlich **while**-Schleifen zur  
wiederholten Ausführung nutzt.

...und stärker:

- ▶ Jedes **W**HILE-Programm lässt sich in ein bedeutungs-  
gleiches **W**HILE-Programm mit **einer einzigen while**-  
Schleife überführen, ein Programm in sog. **Engelerscher**  
**Normalform**.

Nicht immer jedoch ist dies so einfach wie für

- ▶ **repetitive** und **lineare Rekursion**.

# Übungsaufgabe E.1

Löse die Probleme

1. Türme von Hanoi
2. fun91

nach dem Vorbild der **Haskell**-Funktionen **hanoi** und **fun91** rekursiv in einer **imperativen** oder **objektorientierten** Sprache, z.B. **Java**.

```
hanoi :: Turmhoehe -> A_Stapel -> Z_Stapel -> H_Stapel
      -> [(Scheibe,Von,Nach)]

hanoi n a z h
  | n == 0      = []
  | otherwise =
    (hanoi (n-1) a h z) ++ [(n,a,z)] ++ (hanoi (n-1) h z a)

fun91 :: Integer -> Integer
fun91 n
  | n > 100 = n - 10
  | n <= 100 = fun91 (fun91 (n+11))
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe E.2

Löse die Probleme

1. Türme von Hanoi
2. fun91

iterativ, d.h. gib für die Haskell-Funktionen `hanoi` und `fun91` gleichbedeutende Programme der Sprache `WHILE` oder des `while`-Kerns von z.B. `Java` an:

```
hanoi :: Turmhoehe -> A_Stapel -> Z_Stapel -> H_Stapel  
      -> [(Scheibe,Von,Nach)]
```

```
hanoi n a z h  
  | n == 0      = []  
  | otherwise =  
    (hanoi (n-1) a h z) ++ [(n,a,z)] ++ (hanoi (n-1) h z a)
```

```
fun91 :: Integer -> Integer
```

```
fun91 n  
  | n > 100 = n - 10  
  | n <= 100 = fun91 (fun91 (n+11))
```

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Übungsaufgabe E.3

Gib für die **iterativen** Programme aus **Übungsaufgabe E.2** gleichbedeutende Programme mit je nur **einer einzigen while-Schleife** an.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

Was man kann, erfährt man  
nur durch eine Prüfung.

Seneca der Jüngere (um 4 v.Chr. - 65 n.Chr.)  
röm. Politiker, Philosoph und Schriftsteller  
De providentia 4,3

F

Hinweise zur schriftlichen Prüfung

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Hinweise zur schriftlichen LVA-Prüfung (1)

## ► Worüber:

- Vorlesungs- und Übungsstoff.
- Folgende zusammengehörende wissenschaftliche Artikel:
  1. Konrad Hinsin. [The Promises of Functional Programming](#). Computing in Science and Engineering 11(4): 86-90, 2009.
  2. Konstantin Läufer, George K. Thiruvathukal. [The Promises of Typed, Pure, and Lazy Functional Programming: Part II](#). Computing in Science and Engineering 11(5): 68-75, 2009.

(zugänglich aus TUW-Netz in IEEE Digital Library)

## ► Wann, wo, wie lange:

- **Haupttermin:** Vorauss. am
  - **Do, den 16.01.2020**, 16:00 - 18:00 Uhr, Hörsaal EI 7 (plus ggf. EI 3 als Überlaufraum), Gußhausstr. 25-29; die Dauer beträgt 90 Minuten.

## ► Hilfsmittel: **Keine.**

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13



# Hinweise zur schriftlichen LVA-Prüfung (2)

- ▶ Anmeldung:
  - ▶ Ist **erforderlich!**
    - ▶ Wann: Von vorauss. **Mo, 09.12.2019 (01:00 Uhr)** bis vorauss. **Mo, 13.01.2020 (12:00 Uhr)**.
    - ▶ Wie: Elektronisch über **TISS**.
- ▶ Mitzubringen:
  - ▶ **Studierendenausweis, Stift** (Papier wird gestellt).
- ▶ Voraussetzung:
  - ▶ Mindestens **50% der Punkte** aus dem Übungsteil.
- ▶ Wichtig:
  - ▶ Verbindlich sind im Zweifel allein die in **TISS** angegebenen Termine, Fristen und Räume.

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13

# Hinweise zur schriftlichen LVA-Prüfung (3)

- ▶ Neben dem Haupttermin wird es drei Nebentermine für die schriftliche LVA-Prüfung geben, und zwar:
  - ▶ zu Anfang
  - ▶ in der Mitte
  - ▶ am Ende

der Vorlesungszeit im SS 2020. Zeugnisausstellung stets zum frühestmöglichen Zeitpunkt; insbesondere nach jedem Klausurantritt; spätestens nach Ablauf des letzten Termins für die schriftliche Prüfung.

- ▶ Auch zur Teilnahme an der schriftlichen LVA-Prüfung an einem der Nebentermine ist eine Anmeldung in TISS zwingend erforderlich.
- ▶ Die genauen Termine sind in TISS angekündigt!

Vergiss nicht: Erfolg ist die  
Belohnung für schwere Arbeit.

Sophokles (497/496 - 406/405 v.Chr)  
griech. Dichter

Es war sehr schön,  
es hat mich sehr gefreut.

Franz-Josef I. (1830-1916)  
Kaiser von Österreich

Inhalt

Teil I

Kap. 1

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Teil III

Kap. 7

Kap. 8

Kap. 9

Teil IV

Kap. 10

Kap. 11

Teil V

Kap. 12

Kap. 13